The Role of Knowledge Modeling Techniques in Software Development: A General Approach Based on a Knowledge Management Tool

Jose Cuena and Martin Molina

Department of Artificial Intelligence, Technical University of Madrid

Campus de Montegancedo s/n, Boadilla del Monte, 28660-Madrid (SPAIN)

{jcuena,mmolina}@dia.fi.upm.es

http://www.dia.fi.upm.es

This version corresponds to a preprint
of the actual paper published in:
International Journal of Human and Computer Studies,
(2000) 52, 385-421

Abstract. The aim of the paper is to discuss the use of knowledge models to formulate general applications. First, the paper presents the recent evolution of the software field where increasing attention is paid to conceptual modeling. Then, the current state of knowledge modeling techniques is described where increased reliability is available through the modern knowledge acquisition techniques and supporting tools. The KSM (Knowledge Structure Manager) tool is described next. First, the concept of knowledge area is introduced as a building block where methods to perform a collection of tasks are included together with the bodies of knowledge providing the basic methods to perform the basic tasks. Then, the CONCEL language to define vocabularies of domains and the LINK language for methods formulation are introduced. Finally, the object oriented implementation of a knowledge area is described and a general methodology for application design and maintenance supported by KSM is proposed. To illustrate the concepts and methods, an example of system for intelligent traffic management in a road network is described. This example is followed by a proposal of generalization for reuse of the resulting architecture. Finally, some concluding comments are proposed about the feasibility of using the knowledge modeling tools and methods for general application design.

1. INTRODUCTION

The use of knowledge based systems has been limited to an area of very specific applications where special methodologies and tools are used (different from the techniques applied for software engineering) oriented to model, according to different conventions of knowledge representation, the expertise in several commercially relevant fields. The Software Engineering field has been mainly focused in information systems development improving the reliability and efficiency of data services. However, the current users of these services are

being increasingly interested in deeper functions integrated in the information systems supported by the knowledge related with the data conceptual domains.

The relationship between both approaches has been produced only in the area of knowledge based support to software engineering tasks. Lowry, Duran, (1989) summarize this recent evolution in two main trends: (1) improvement of automatic program synthesis techniques aiming to transform in operative programs high level specifications using set theory and logic such as the commercial system REFINE or the experimental system KIDS Smith (1988) built on top of REFINE, and (2) broadening the automatic programming scope to the entire software life cycle by building knowledge based assistants for acquiring validating and maintaining specifications. These capacities have been embedded in CASE tools. However, three circumstances are creating a different situation:

- The need for a more open architecture in applications to ensure an *adequate human-machine interaction* according to the recent approaches for design that follow a user-centred view.

- The need of *software reuse* which requires an open structure: (1) to easily understand the contents of any software component and (2) to be able of accepting changes in its contents according to the specific needs of the application where the component is going to be reused.

- The *improvements on reliability and capacity of representation* produced in the last ten years in the field of knowledge representation and knowledge acquisition methods, giving

birth to a collection of mature technologies supported by experimental tools, yet, but providing levels of services very close to the industrial requirements.

Therefore, now it is possible to formulate and to build an application by using directly the knowledge modeling concepts supported by adequate tools instead of formulating the application using information structuring concepts and data processing algorithms as in the usual software environments. This is a very important feature because it approaches the design phase to the conceptual specification phase, that usually in the traditional software world are separated by a bigger gap and, hence, subject to more errors than the errors possible between the conceptual model and the knowledge model which is closer to the conceptual abstractions.

However, not many attempts have been produced by the AI community to produce something like cognitive programming environments in an operational way where reasoning steps using domain models are applicable to describe and to explain the answers of the application. AI has to invade with practical views the area of applications. Although the paradigm modeling efforts must continue as focus of research, an additional focus should be the advanced modeling of complex applications using the available paradigms.

This paper aims precisely to propose a type of this structured knowledge model formulation based on a tool oriented to support the design and implementation of general applications using the knowledge engineering approach which means to understand the current applications from a richer conceptual perspective. The interest of the paper is to provide some initial results on the possibilities of this class of tools to be acceptable by the general applications development community.

First, a summary of the actual requirements for software development is commented. Then, the concept, the structure and the organization of the KSM (Knowledge Structure Manager) tool is presented conceived to support and extend the state of the art in knowledge modeling approach. Then, an application using KSM for real time emergency management is described where practical comments are included. Finally, some general conclusions are proposed about the role of the knowledge oriented approach in the context of Software Engineering by evaluating the behavior of the model experimented with respect to the usual parameters and criteria applied for software evaluation.

2. GOALS OF THE SOFTWARE DEVELOPMENT SUPPORT

The conventional software field has evolved after the first crisis of software at the end of the sixties in terms of better human understanding models for applications and the supporting programming languages. Thus, now, there exists as main well understood programming paradigms the object oriented ones, based mostly in C++, Java and CLOS, and the logic programming ones based on different versions of Prolog language based environments. The methods for design of applications supporting the evolution of the formulation of concepts from the human mind structure format to the computable format on some programming paradigm have been formalized in structured life cycles where the different steps of requirements analysis, design, implementation, test and maintenance are detailed in diverse standard processes derived from the initial proposals of Yourdon, De Marco, Weinberg, etc. now summarized in the methodologies Metrica, Merise, Ssadm, OMT, etc. Pressman (1992). To support these life cycles, different CASE tools have been proposed guiding the design and maintenance process of an application from the conceptual specifications to computable

models. Increasing attention is given to the research area of Requirements Engineering aiming to the conceptual modeling via specifications of the underlying human understanding of applications. Finally, reuse techniques based on these conceptual modeling approaches have been established without not too innovative results yet with respect to the traditional reuse of libraries of functions and libraries of classes supported by objects. At the current state of the art any environment to support application design and development should provide as main functions:

- Structuring and encapsulation facilities to ensure an adequate size of the different components of the application and an understandable format to allow easy access to the different component modules.

- Software sharing facilities to ensure that no function is formulated twice with the corresponding inconsistency and redundancy risks for operation and maintenance.

- Software reuse potentiality to ensure the use of the already experimented existing applications.

- Advanced Human Computer Interaction support, to ensure adequate and reliable user and programmer contribution to maintenance and operation of the applications.

- An adequate level of efficiency in the operation for the needs of the user.

- Test and validation facilities.

The current state of knowledge modeling technology allows to contribute with quality enough to the previous items by providing higher levels of conceptual modeling in consonance with the growing trend in Requirements Engineering. In the following paragraphs a brief summary of the knowledge modeling area is presented and an example of a product, summarizing state of the art in knowledge acquisition, used for application development is described.

3. GENERAL VIEW OF THE KNOWLEDGE MODELING METHODOLOGY

First generation knowledge-based systems provided a set of standard reasoning procedures using declarative representations (such as rules, frames, etc.). The next generation of knowledge-based systems abstracted from symbolic representational considerations the design process and evolved to the paradigm of model-based system development, in which a knowledge system is viewed as an operational model capable of simulating a certain observed problem solving behaviour from an intelligent agent (e.g., a human expert in a certain professional field). This view contrasts to the traditional approach where a knowledge system was usually considered as a container to be filled with knowledge extracted from an expert. The modelling process considers the existence of an abstract level where the knowledge can be functionally described showing its role in the problem solving process, independently of a particular representation. This level, proposed by Newell with the name of *knowledge level* Newell (1982), allows to describe a knowledge model in terms of strategies of reasoning and roles of knowledge types, abstracting away from how these are implemented by specific symbolic representation formalisms. After some years of different proposals for knowledge modeling at the knowledge level, the knowledge acquisition community agreed several key

concepts such as the *generic task* (proposed by Chandrasekaran (1983, 1986), also present with some variants in the KADS model Wielinga et al. (1992) and in the model of components of expertise Steels (1990)), the *role limiting method* McDermott (1988) and the *ontology* concept Gruber (1993). According to these concepts we can distinguish two main organization principles for structured knowledge based applications:

- *The task oriented principle*. A *task* is defined as a goal to be achieved (for instance diagnosis of infectious diseases or design of the machinery of an elevator). It is described with the type of inputs it gets and the type of outputs it produces. The main function of the model is represented by a global task. This task is decomposed into simpler subtasks developing a tree which shows the general structure of the model. A *problem-solving method* (or *method* in short) defines a way in which the goal of a task can be achieved through the execution of subtasks, so that when a method is associated to a task, the method establishes how the task is divided into subtasks. Thus, a knowledge model can be understood as a hierarchical composition of tasks where each task is carried out by a problem-solving method. We may call this organizational principle the *task-oriented* organization that makes emphasis in *procedural* knowledge given that it mainly shows *how* to reason for solving problems integrating other, simpler, problem solver results.

- *The domain oriented principle.* On the other hand, the notion of *ontology* was defined to describe explicit specifications of domain elements. An ontology is a declarative description of the structure of a particular domain. The use of ontologies allows to more easily reuse and share knowledge about certain domains to carry out different tasks. This organizational principle makes more emphasis in *declarative* knowledge.

These principles need to be combined adequately in order to formulate a knowledge based application. A reasonable approach to be followed in the design process is to start from the collection of *top-level tasks* that describe the set of goals to be achieved by the application to support an adequate conversation model between the user and the system. These top-level tasks may be the basic types of answers required in such a conversation model. For each top-level task, a hierarchical structure of task-method-domain may be used to show the way the final task supports an answer type (Figure 1). Each hierarchy represents how each task is carried out by a specific method, decomposing the task into simpler subtasks. Usually, the hierarchy will present only one method associated to a task. However, in the near future, when reusable libraries of problem-solving methods will be available, it could be more usual to associate more than one method to a task, developing a more complex architecture (this architecture could be named the *problem solving medium*). This means that the same task will be able to be solved in different ways depending of certain dynamic characteristics (such as the type of dialogue with the user, the context, etc.). At the bottom level of the hierarchy of task-method-subtasks there is a collection of primary methods associated to primary tasks. What is considered as a primary method is a design decision established by the developer. Typically, primary methods correspond to methods that can be directly implemented at symbolic level by simple problem-solving techniques (such as knowledge based techniques like backward or forward chaining in rule-based representations, network-based representations, constraint satisfaction methods, and also specific algorithmic solutions that do not require a explicit representation of declarative knowledge).

The use of declarative knowledge by primary methods requires an ontological definition of such a knowledge that is viewed as a set of domain models that support primary tasks. Domain models can be formulated with two components: (1) a *conceptual vocabulary* where

a concept-attribute format organized in classes and instances may be used to establish the basic language of the domain model, and (2) relations between these concepts described by the corresponding declarative *knowledge base* (that at symbolic level will be formulated as frames, constraints, rules, etc.). The same vocabulary and knowledge base of a domain model may be used by several methods (showing cases where the same concepts play several roles) and also a subtask can be part of different methods.

As an example of the previous ideas, consider a simplified generic model for decision support for management of a dynamic system (e.g., a chemical plant, a traffic network, etc.) Cuena, Hernández (1997). The goal of this decision support model is to help an operator in detecting and diagnosing problems in the dynamic system, as well as to help in choosing appropriate regulation actions to cope with the detected problems. Figure 2 shows a possible task-method-domain structure for this case. Four classes of questions are considered in this model: *what is happening, why is it happening, what may happen if* and *what to do if*. These questions correspond to the four top level tasks: *classification, diagnosis, prediction* and *configuration*. Each top-level task is carried out by a particular problem-solving method. For instance, the diagnosis task is carried out by the method model-based diagnosis, which decomposes the tasks into two simpler tasks: *propose causes, filter causes*. At the bottom level, there are primary methods such as *qualitative abstraction, pattern matching, instantiation,* etc. Each primary method uses a particular declarative model. For instance, the method *causal covering* uses the declarative model *causal model*. Note that certain declarative models may be used by several methods (e.g., in this example, the system structure model).

Thus, following this approach, a cognitive architecture is formulated as a collection of task-method-domain hierarchies for each of the basic questions to be answered through the user

interface of the system. However, in real applications, the experience shows that, sometimes, too large descriptions can be produced by using this type of formulation. Thus, although the conceptual description based on task-methods-domains is adequate for the analysis process, it needs to be complemented and re-organized using additional modelling concepts for the final design of the application. There are some reasons for this: (1) the task-method-domain structure presents too much level of detail that increases the difficulties to understand and maintain complex architectures and (2) the level of disaggregation of components can produce software implementations with problems of efficiency.

Therefore, as it happens in the conventional software field, it is required to have a synthetic view of an application at several levels of conceptual aggregation allowing easy understanding and, hence, easy maintenance. This is an issue not too much considered by AI research community who is mostly focused in the identification of innovative models and paradigms but not so much in final tools supporting design and maintenance of applications. This type of tools are more common in software engineering (e.g., CASE environments) that support conventional methodologies (such as DDF, OMT, etc.) together with libraries of software components that can help in the design and implementation process of an application. However, these tools are based on the traditional perception of applications as data+algorithms that is not enough to be used in some applications that require also knowledge-based solutions. Thus, from the point of view of knowledge modelling, several preliminary proposals have been produced (e.g., SHELLEY or MIKE Landes, Studer, (1994) that follow the KADS methodology, or PROTEGE-II Musen, Tu (1993) and KREST McIntyre (1993)).

4. KSM: A KNOWLEDGE MODELING TOOL

A proposal in the direction commented in the previous paragraph is the KSM (Knowledge Structure Manager) environment. KSM follows the described task-method-domain approach but it introduces some new description entities that facilitate the design process of the application.

The main structuring concept in KSM is what is called *knowledge area*[1] which is a block summarizing parts of the global structure of task method subtasks. A knowledge area in KSM follows the intuition of body of knowledge that explains a certain problem solving behaviour. A cognitive architecture that models the expertise of a professional can be viewed as a hierarchically structured collection of knowledge areas at different levels of detail, where each knowledge area represents a particular qualification or speciality that supports particular problem solving actions which appear in the global task method subtask structure. Thus, each module that represents a knowledge area, in contrast to the functional view that provides a task (which is an answer the question *what does it do),* is an answer to the question *what does it know* at different levels of detail. The concept of knowledge area in KSM is useful as a basic module for structuring tasks, methods and domains. A knowledge area (figure 3) is described with two well differentiated parts: (1) its knowledge, represented as a set of component sub-areas of knowledge, and (2) its functionality, represented by a set of tasks (and their corresponding methods). The first part decomposes the knowledge area into simpler subareas, developing a hierarchy at different degrees of detail. The second part associates tasks to knowledge areas showing their functional capabilities.

The knowledge area concept is useful to produce a more synthetic view of the knowledge model given that it groups a set of tasks (together with the corresponding methods) in a conceptual entity of higher level. Figure 4 shows this idea. The tasks of the task-method-domain hierarchy on the left can be grouped in five knowledge areas. In this process, the developer must follow the rule that a method corresponding to a task T of a certain knowledge area A only can use subtasks provided by the subareas of the area A. In principle, given a hierarchy of task-method-domain resulting from the knowledge level analysis, it is possible to design different hierarchies of areas according to a principle of knowledge area structuring. In order to guarantee a reasonable level of understandability of the knowledge model, the final hierarchy should be designed to follow the natural intuitions associated to the knowledge attibuted to the human problem solving process to be modeled.

Figure 5 shows a possible structure of knowledge areas corresponding to the previous example (figure 2) where different knowledge areas embody the different methods and its corresponding domain knowledge. For instance, in the example, a top-level knowledge area, called *decision support knowledge*, has been designed to include as top-level task the decision support one integrating as component subtasks, provided by its component knowledge, *classification*, *diagnosis*, *prediction* and *configuration* required to answer the three main question types (*what is happening, why is it happening, what may happen if* and *what to do if*). This area makes use of other three intuitive knowledge areas: *problem knowledge*, *behaviour knowledge* and *regulation knowledge*. The bottom-level knowledge areas correspond to what is called *primary areas*, that include one single declarative model together with the set of tasks that make use of such a model. For instance, the area called *patterns of problems* is an example of primary area and includes a declarative model with a

set of problem patterns together with the task *match* that receives a set of facts corresponding to a current state of the system and finds problem patterns that satisfy their conditions.

Knowledge areas can be defined at generic and at domain level. Generic areas mean classes of bodies of knowledge that allow to formulate a model. Then, a particular domain model is viewed as a collection of instances of such classes that can share by inheritance different properties of the classes such as relations with other areas, problem-solving methods, etc. This possibility of defining classes of areas is a solution to support reuse. Thus, abstract structures of knowledge areas may be reused to develop different applications operating in different domains.

The formulation using the knowledge area format provides certain advantages: (1) every task at any level of the hierarchy is associated to the explicit knowledge that supports its functionality, which makes more meaningful the model (2) the structure of knowledge areas synthesizes the structure of tasks-method-domains, which is useful to better understand complex models, (3) at the bottom level, primary knowledge areas encapsulate declarative domain models, so it is a solution to organize the domain layer in separate modules, which contributes to keep easier the consistency of the model and (4) primary knowledge areas are easy to be implemented by reusable and efficient software components, which gives a solution for the development and maintenance of the final executable version of the system.

This structuration contrasts to a plain organization of knowledge, such as the traditional structure proposed in the original rule-based systems that does not describe explicitly the different knowledge modules in which rules could be organized. Knowledge areas allow to identify such modules, even by establishing several conceptual levels (knowledge areas being

part of other knowledge areas). Thus, the resulting system can describe better its own knowledge (more similar to how a human expert does) showing the categories in which it can be classified. This contributes to present different levels of detail of the expertise, and allows to produce good quality of the explanations which may be poduced by tracing the reasoning steps at different levels.

The organizational principle followed in KSM may be called *the knowledge-area oriented principle*. In order to summarize and compare the organizational principles mentioned in this paper, it may be established an analogy with similar principles followed in software engineering:

- The task oriented principle is somehow similar to the functional description used in the top-down methodology of structured analysis where a process is systematically decomposed into simpler processes developing a hierarchy. However, tasks in contrast to the traditional processes, are not viewed as procedures for data processing, but are considered reasoning steps within a global problem solving behaviour observed in a human expert; every reason step uses a body of knowledge available in the component knowledge areas (for instance, in the primary areas a reasoning step uses a frame or a rule to evaluate if the class modelled by the frame is true or to chain the rule with the current state in the working memory).

- The domain oriented principle presents similarities to the data-base design where data have to be organized according to a particular scheme. The domain oriented principle however is established at a more abstract level and also consider more complex declarative organizations than the ones usually supported by conventional databases.

- The knowledge-area oriented principle is somehow similar to the object oriented principle that encapsulates in intuitive entities processes and data. Knowledge areas, however, are associated to the intuition of a body of knowledge, which gives a more specific semantics to this component and it is useful to naturally explains a possible set of cognitive skills that justifies the problem solving competence of a human expert in terms of a collection of associated methods to perform several tasks.

This three modelling principles are very useful to analyze the expertise in a particular domain problem and to develop a formal design that allows the construction of an operative model on the computer. They provide a new logical level for system conceptualization, closer to human natural intuitions and, therefore, easier to be understood by non computer scientists. The proposed description entities follow cognitive metaphors which allow to have a more natural perception of the resulting application. It is important to note that these principles are very general and can be used to different kind of problem-solving applications, i.e., they are useful for both knowledge based and conventional applications, providing a unified view for development of applications.

4.1. The KSM Languages to Formulate a Knowledge Model

KSM provides two formal languages to formulate two characteristics of the model: common terminologies about the domain (conceptual vocabularies) and strategies of reasoning (problem solving methods). Both languages are used by the developer to refine the knowledge

model that previously has been defined as a structured collection of knowledge areas together with tasks and methods. These two languages are the Concel language for vocabularies, and the Link language for problem solving methods. This section explains both languages.

4.1.1. A Language for Vocabularies: The Concel Language

The declarative description of a domain within a model can be viewed as a collection of classes concepts, relations, structures, etc. In order to facilitate an efficient operationalization of the final model, it is important to distinguish between the domain descriptions that are common to the whole model and additional extensions oriented to perform specific primary tasks. In KSM, the common descriptions are formulated with what is called *conceptual vocabularies* and the extensions are written within specific knowledge bases using different symbolic representations. This section describes the language used in KSM to formulate conceptual vocabularies. Section 3.3 explains how to write additional descriptions of the domain oriented to carry out particular tasks, using specific symbolic representations taken from a library of primitives of representation.

A conceptual vocabulary allows the developer to define a common terminology which can be used by different primary knowledge areas. One of the direct advantages of the use of vocabularies is that they provide a common location where concepts are defined. This avoids to repeatedly define the same concepts eliminating the risk of incoherence in the knowledge of different domains The concepts defined by the vocabulary will be later referred by other

symbolic representations (rules, frames, constraints, etc.) used by primary areas. Due to the general use of vocabularies by different knowledge modules, they must be formulated in a common language. KSM provides the Concel language for this purpose. It allows the developer to define: concepts, attributes, and facet values and the classification of the concepts in classes and instances.

In more detail Concel uses the following elements. The basic element is the *concept*. Examples of concepts are: a sensor, a symptom, a disease, etc. Each concept has *attributes* which describe characteristics of the concept. For example, the concept gas may have the following attributes: pressure, volume and temperature. Each attribute has also its characterization through *facets*. Concepts can be organized into classes and instances. A *class* concept represents a family concepts. For example, the class concept sensor represents the generic concept of the sensor family. The elements of a family are called *instances*. For example, S0735 is a instance of sensor. The general syntax to define concept is:

According to this format, each concept is defined with a name. It can be either a subclass of a higher level class or an instance of a class. The concept can be described with a collection of attributes and each attribute is defined with a collection of facets. The possible facets are:

- *Type integer*. It defines that the attribute has integer values. Optionally a range can be defined to establish the limits. The formulation of this facet is (INTEGER [RANGE <min> <max>]). For instance (INTEGER RANGE 125 235).

- *Type interval.* The attribute has as possible values numerical intervals. Optionally a range can be defined to establish limits. The format of this facet is (INTERVAL [RANGE <min> <max>]). For example (INTERVAL RANGE 0 200).

- *Type boolean*. The attribute may have one of the two boolean values true and false. It is written with the format (BOOLEAN).

- *Type instance.* The values of the attribute are instances of a class. The format is (INSTANCE OF <class>). For example (INSTANCE OF symptom).

- *Type qualitative values.* The values of the attribute are defined as a list of possible qualitative values. The format is {<value-1>, <value-2>,... , <value-n>}. For example {low, medium, high}.

- *Default value.* It defines a constant value for the attribute. The value is written after a colon, following the format : <value>. For instance: 5.

- *Units*. It defines the units in which the attribute is measured. The unit is defined between brackets with the format [<unit>]. For example [minutes].

The following example illustrates a complete definition of a class:

```
CONCEPT Urban Section SUBCLASS OF Section.
ATTRIBUTES:
            Capacity       (INTERVAL RANGE 0 2000) [Veh_Km],
            Lanes          (INTEGER RANGE 1 4): 1,
```

```
Detectors      (INSTANCES OF Detector),
Length         (INTEGER RANGE 0 1000) [m],
Speed          {low, medium, high},
Circulation    {free, saturated, congested}.
```

This example defines the class called urban section as a subclass of the concept section. It is defined with six attributes where there are both numerical and qualitative attributes. For instance the attributes *lanes* and *length* are integers (with ranges 1-4 and 0-1000 respectively) and the attribute *capacity* is an interval (with range 0-2000). There is a default value for the attribute lanes (one lane). The attributes *capacity* and *length* have units (vehicles/Km and meters respectively). On the other hand the attributes *detectors*, *speed* and *circulation* have qualitative values. In the case of *speed* and *circulation* they present explicitly the set of possible values (e.g., low, medium and high for speed). The type of values of the attribute detectors are defined as instances of the class detector. The following example shows a case of the definition of an instance:

```
CONCEPT Main Street IS A Urban Section.
ATTRIBUTES:
              Capacity:     [1400, 1800] [Veh_Km],
              Lanes:        3,
              Detectors:    (DE1003, DE1005),
              Length:       350 [m].
```

This example defines the concept *main street* as an instance of the class *urban section*. In this case, particular values are associated to some attributes defined in the class. A generic model include conceptual vocabularies that define normally classes of concepts (and possibly also instances) that are domain-independent. The particular instances or subclasses of such

concepts corresponding to a specific domain will be defined later when the model is instantiated on such a domain.

4.1.2. A Language for Problem-solving Methods: The Link Language

In order to describe how a task is carried out, a developer defines a method with a particular problem-solving strategy. Methods may be considered control knowledge given that they describe control strategies about the use of domain knowledge. They formulate how the system reasons when it solves a problem; in other words, they formally define the problem-solving behaviour of the knowledge model (from a different point of view, considering the knowledge modeling activity as a process of selecting, adapting and assembling reusable building blocks, the method formulation may be considered also as a process of linking knowledge components to construct the whole knowledge model).

Basically, using the Link language, the method formulation includes on the one hand, the data connection among subtasks and, on the other hand, the execution order of subtasks (a deeper description of the Link language can be found at Molina et al. (1998a)). The view of each particular subtask to be used by a method is divided into two levels (the data level and the control level). The *data level* shows input data and output data. For instance, the task of classification receives as input measures and generates as output a category. Likewise, the task of medical diagnosis receives as inputs symptoms and the case history of a patient and generates as output a disease and a therapy. On the other hand, the *control level* offers a higher level view of the tasks showing an external view about how the task works. This level includes two elements of information: control parameters and control states. A control

parameter selects how the task must work when it accepts different execution modes. For instance, a classification task classifies into categories measures received as input data according to a similarity degree. The similarity degree may be considered as a control parameter. In the context of a real time system, other examples are the maximum reasoning time or the maximum number or answers, when more than one could be expected. Control states, in their turn, indicate the degree of success or failure of the task after the reasoning. For instance, the medical diagnosis task may have as possible control states: insufficient data (when there are not enough data to give a result), healthy patient (when the patient does not have any disease), no therapy found (when the patient has a disease but the system does not find out a therapy) or therapy found (when the patient has a disease and the system finds out a therapy). Note that control states do not provide the actual results of the task, but they give an abstract information about how the tasks worked. In summary, at the control level of a task, control parameters selecting modes are received as input and control states informing about the reasoning are generated as output.

According to this division, the formulation of a method using the Link language includes several sections (figure 7). After the name of the method, the first section, that is called *arguments*, indicates the global inputs and outputs of the method. Then, there are two main sections: the data flow and the control flow. The *data flow section* describes the data connection of subtasks at the data level, indicating how some outputs of a task are inputs of other tasks. The *control flow section* describes the execution order of subtasks using control rules that include control states and parameters. In addition, there are also other two optional sections: the control tasks and the parameters. The *control tasks section* allows the developer to include tasks that decide the execution of other tasks, and the *parameters section* is used to write default values for control parameters.

The *data flo*w section describes the data connection of subtasks showing how some outputs of a subtask are inputs of other subtasks. The developer here writes input/output specifications of subtasks using what is called flow. A *flow* identifies a dynamic collection of data, for instance the symptoms of a patient in medical diagnosis or the resulting design of an elevator. For a given method, there are several names of variables identifying the different flows that will be used to connect subtasks. These variables represent plain flows, i.e. flows whose internal organization is not known at this level. In addition, complex flows, called flow expressions, can be written as the composition of others using a set of basic operators (conjunction, disjunction, selection, list, etc.). To formulate this inference structure, the developer writes a collection of input/output specifications (i/o specifications). Each i/o specification includes, first, the subtask name as a pair made of the knowledge area name and the subtask identifier. Second, it is defined the input of the subtask. Basically, the input is defined with names identifying flows (plain or complex flows). Each input flow accepts a *mode* that may be the default mode or the one-of mode (the default mode gets all the elements of a list at once, while the *one-of* mode gets element by element, which is useful to formulate non-deterministic search methods). Finally, the output is defined with a list of single identifiers giving names to the output flows. In Link language, in general, subtasks are considered non-deterministic processes. This means that as a result of a reasoning, a task may generate not just one result, but several ones. For instance, in the context of medical diagnosis, the task may deduce several diseases and several therapies for the same symptoms. So, when tasks are going to be connected in the data flow section this possibility must be taken into account. This is managed with two output modes. Modes select whether the whole set of outputs must be generated one by one element considering that there is a non-deterministic result (this is the default mode) or, on the contrary, it must generate all the outputs at once as a list of single elements for each output flow, which is called the *all* mode.

The purpose of the *control flow* section is to provide a formal description of a control strategy that determines the execution order of subtasks. The representation uses production rules for the control flow. The advantage of this representation is that it easily may define local search spaces considering the non-deterministic behavior of subtasks. At the same time, the representation is simple enough to be used easily due to this language is not a complex programming language but, on the contrary, it was designed to serve as an easy description to formulate procedural knowledge (a method will have a small number of rules, usually less than 10). Using production rules provides a intuitive representation, and flexibility for maintenance. The format of a rule is: (1) the left hand side includes a set of conditions about intermediate *state of task executions* is, and (2) the right hand side includes a sequence of *specification of task execution* . Each one of the first elements (state of task executions) is a triplet *<K,T,S>* where *K* is a knowledge area, *T* is a task identifier and *S* is a control state. This means that the result of the execution of the task *T* of the area *K* has generated the control state *S*. The value of *S* is control information such as successful execution or failure of different types, which may be used as premises to trigger other production rules. The representation of the elements in the RHS (specification of task execution) is another triplet *<K,T,M>*, where *K* is a knowledge unit, *T* a task and *M* an execution mode. This representation means that the task *T* of the knowledge unit *K* must be executed with the execution mode *M*. The execution mode expresses the conditions limiting the search such as: maximum number of answers allowed, threshold for matching degree in a primary unit using frame representation, time-out, etc. For instance, the following rule is an example of this representation:

```
<K: validity, T: establish, S: established>,
<K: taxonomy, T: refine, S: intermediate>
->
```

<K: taxonomy, T: refine, M: maximum 3 answers>

<K: validity, T: establish, M: null>.

However, in Link language, this representation has been modified to include some syntactic improvements. A complete example of a method formulation for hierarchical classification using the *establish-and-refine* strategy is presented below, where the second rule within the control flow section correspond to the previous rule but re-written according to the syntax of Link:

```
METHOD establish and refine
ARGUMENTS
       INPUT description
       OUTPUT category
DATA FLOW
       (validity) establish
                     INPUT description, hypothesis
                     OUTPUT category
       (taxonomy) refine
```

```
                    INPUT category
                    OUTPUT hypothesis
CONTROL FLOW
       START
       ->  (taxonomy) refine, MODE maximum answers=3,
           (validity) establish.

       (validity) establish IS established,
       (taxonomy) refine IS intermediate hypothesis
       ->  (taxonomy) refine MODE maximum answers=3,
           (validity) establish.

       (validity) establish IS established,
       (taxonomy) refine IS final hypothesis
       -> END.
```

The representation also includes references to the beginning and the end of the execution to indicate the first set of actions to be done and when it is considered that the process has reached a solution of the problem. The beginning of the execution is referred as a state of the execution (to be included in the left hand side of the rules) and it is written with the reserved word START. The end of the execution is considered as an action (to be included in the right hand side of the rules). It is written with the reserved word END and, optionally, can be followed by a symbol that expresses the control state that has been reached.

In addition to the previous representation, the Link language includes also the possibility of formulating a more complex control mechanism by using what is called *control tasks*. These tasks are included in the *control task* section in the same way that is formulated in the *data flow* section. The main difference is that control tasks produce as output, instead of only flows (at data level), tasks to be executed, formulated as task specifications. These task specifications can be included in the right hand side of control rules to determine when they

must be executed. In addition, a control task can get as input the execution state of another tasks. In this way, it is possible to build models that include specific knowledge bases that include criteria to select the next tasks according to the execution of previous tasks. These solutions provide the required freedom to use the most appropriate knowledge representation and inference for different control strategies. Another utility of the use of control tasks is that they make possible to implement a dynamic selection of methods for tasks. The idea is that a control task uses a knowledge base that establish how to select the most appropriate method and, as output, the control tasks generates the name of a subtask (with the corresponding method) to be executed.

Concerning the execution of a method formulated using the Link language, it follows the control established by the set of control rules. In the simplest case, when this sequence is previously known and it is permanent, there is just one rule with the explicit order at the right hand side. However, the use of control rules allows to define more complex situations. First, it allows to dynamically determine the sequence of execution, so that it is possible to represent control structures such as *if-then, loops, repeat,* etc. In order to do so, control states are used. For instance, in the previous example of method that follows the establish and refine strategy, the second rule can be triggered in a loop until the hypothesis is not intermediate. In addition to that, in Link language is possible to define a more powerful execution with a non-linear sequence. This is possible by two reasons: on the one hand, for a given state more than one rule may be used and, on the other hand, a given task may generate more than one result. This possibility of non-linear executions is a powerful technique that allows the developer to define more easily problem-solving strategies where there are search procedures. The developer can also modify the search control strategy using some tools provided by Link: input modes (one-of or set), output mode (all, one-each-time), and search parameters (such as

maximum number of replies and time-out). According to this, Link develops a local search space for the execution of a particular problem-solving method. In general, given that a method calls subtasks, each one with its particular method, different local search spaces are developed at run time by the Link interpreter, each one for each method. A concrete example of an execution corresponding to the *establish-and-refine* problem solving method which has been presented in Link language previously is presented in figure 9.

4.2. Primitives of Representation to Operationalize the Knowledge Model

During the development of a particular knowledge model, the developer initially defines an implementation-independent abstract model that constitutes a description of a cognitive architecture. As it was presented, the central structure of this model is defined as a hierarchy of knowledge areas, where each area is divided into subareas until elementary areas are reached (called *primary* knowledge areas).This structure is refined by using the Concel and Link languages. In order to produce the final operational version of this knowledge model, KSM provides a set of software components called primitives of representation. A deeper description of this type of components can be found at Molina et al. (1999).

The purpose of a *primitive of representation* is to provide a symbolic representation together with a set of primitive inference methods to be used in the operationalization of a primary area of a knowledge model. For each primary knowledge area of the model, the developer selects the most appropriate primitive that acts like a template to be filled using domain knowledge in order to create the final operational component that implements the primary area. It is important to note here that the use of primitives of representation (taken from an

open library of primitives in KSM) provides the required freedom to the developer to use the most appropriate representation and inference for each case, which is especially important to ensure the adequate level of efficiency of the final implementation. As a consequence, the declarative description of the domain of a final model will be formulated using different languages, part of it using the Concel language (the common terminology) and the rest written in different languages provided by primitive of representation.

A primitive of representation is a reusable pre-programmed software component that implements a generic technique for solving certain classes of problems. The primitive defines a particular domain representation using a declarative language together with several inference procedures that provide problem-solving competence. In a simplified way, the structure of the primitive is defined by a pair <*L*, *I*>, where *L* is a formal language for knowledge representation and $I = \{i_j\}$ is the set of inferences, i.e., a collection of inference procedures that use the declarative representation written in *L*. The module defined by a primitive is a design decision that is mainly influenced by the representation language *L*. This language is usually homogeneous, declarative and close to personal intuitions or professional fields. In a particular primitive, this language can adopt one of the representations used in knowledge engineering such as: rules, constraints, frames, logic, uncertainty (fuzzy logic, belief networks, etc.), temporal or spatial representations, etc. Also other parameterised or conventional representations can be considered, such as the parameters of a simulator or a graph-based language. Each element of the set of inferences *I* expresses an inference procedure that uses the knowledge formulated in the language *L*. For instance, the rule-based primitive may have an inference, called *forward chaining*, that implements an inference procedure following a forward chaining strategy to determine whether a goal can be deduced from a set of facts given the rules of the knowledge base. In addition, there may be also

another inference that follows the backward chaining strategy for the same goal. Each inference *ij* defines a pair <*P*, *C*> where *P* is a set of inputs (premises) and *C* is a set of outputs (conclusions).

The primitive provides an interesting level of generality due to the abstraction of the domain knowledge that provides the use of the representation language. The same primitive can be used to construct different modules with different domain knowledge. For instance, a rule-based primitive can be used to construct a module to diagnose infectious diseases or it can be used to build a module that classifies sensor data. Both modules are supported by the same primitive but they include different domain knowledge. Another interesting advantage provided by the primitive is that there is a clear analogy between primitives and knowledge areas, so this offers an easy transition from the implementation-independent model (as a result of analysis phase) to a more refined model where elementary computable components have been selected to configure the operational version. This continuity preserves the structure defined by the abstract model and, as a consequence, improves the understandability and flexibility of the final system.

Primitives of representation are combined to develop a complex architecture, following a model defined as a structure of knowledge areas modeling an understanding structure at the knowledge level. Each primitive is associated to one or more primary areas and then, each primary area is part of higher level knowledge areas. Basic tasks provided by primitives are combined to define strategies of reasoning by using the Link language. On the other hand, the representation language of the primitive is used to formulate a declarative model of the domain knowledge. However, this local information could be shared by other different primitives. This problem about common concepts is solved by using of conceptual

vocabularies. Vocabularies define global sets of concepts to be shared by different knowledge areas and, therefore, they have to use a general representation, the Concel language. From the point of view of primitives of representation, they must be capable of sharing vocabularies. The solution to this is that the primitive provides mechanisms to import Concel definitions that are translated to the local representation language of the primitive. Thus, when the user of the primitive needs to write a particular local knowledge base during the knowledge acquisition phase, the vocabularies shared by the primitive are previously imported to be part of the base, in such a way that vocabularies are directly available in the language of the primitive to help in writing the knowledge base.

At the implementation level, the primitive is a software module designed and implemented as a *class* (from the object-oriented development point of view), i.e. programmed with a hidden data structure and with a collection of operations which are activated when the class receives messages. A class implementing a primitive (figure 10) includes, on the one hand, an internal data structure divided into three basic parts: (1) a data structure to support the *local vocabulary* used by the knowledge base (for instance, in the case of a representation of rules, this part contains the set of concepts, attributes and allowed values that will be valid in the knowledge base), (2) a data structure that implements the internal structure that supports the *knowledge base* as a result of the compilation of the language provided by the primitive, and (3) a *working memory* that stores intermediate and final conclusions during the reasoning processes together with traces that can serve to justify conclusions through explanations. The data structures (1) and (2) are created during the knowledge acquisition phase and the data structure (3) is created and modified during the problem-solving phase when inference procedures develop their strategies of reasoning.

On the other hand, the class implementing a primitive includes a set of external operations that can be classified into three types: (1) *knowledge acquisition operations*, whose purpose is to help the user in creating and maintaining the knowledge base, (2) *problem-solving operations* that execute the inferences provided by the primitive; they receive a set of inputs and generate responses using the internal structure representing the knowledge base, and (3) *explanation operations*, that justify the conclusions using the traces stored in working memory. If the primitive is not knowledge based, the corresponding object includes neither knowledge acquisition nor explanation operations.

During the *creation* of a knowledge model, the developer constructs each knowledge area using the corresponding primitive, which is implemented by a particular class. Internally, an instance of the corresponding class is automatically created. Certain operations for knowledge acquisition can be invoked (by inheritance) to construct and modify the knowledge base: import a conceptual vocabulary, edit the knowledge base (using an external user-friendly view of the knowledge base to the operator, with facilities to create and modify) and machine learning procedures. During the *execution* of tasks of the knowledge model, the problem-solving operations of the corresponding objects of the primitives are invoked with input data. Those local operations navigate through the internal data structure of the knowledge base to generate outputs. During the problem solving reasoning, the operations produce intermediate and final conclusions that are stored in the working memory. This information can be used later, when the user of primitives wants to get explanations that justify the conclusions of the reasoning.

In summary, the reusable component for knowledge modelling, the primitive of representation, is implemented by a software object. In fact, object-oriented methodologies,

that have already a long tradition in software engineering, provide a good context for reuse. The philosophy of the object-oriented design proposes a more stable modularity based on the identification of components of a certain world (real or imaginary), instead of the original modularity based on functions or processes that tends to be less stable. This philosophy is adequate for implementing primitives where the intuition associated to each object is the representation technique used by the primitive. The language where each primitive must be formulated is open. Different programming languages such as C,C++, Fortran, Prolog, etc may be applied. If they are knowledge-based they must have a user interface to acquire the structures of representation for the knowledge base (such as rules or frames). This activity is carried out by programmers outside of KSM using particular programming languages. Once a particular primitive is built, it must be individually validated and then it is integrated in the KSM library as a reusable software component. However, for a specific application, it is not always necessary to program the complete set of primitives of representation. The reason for that is that some primitives may exist already in the KSM library. They were developed previously for a different application, so that they can be reused for the development of a new one. Therefore, just part of the primitives will have to be programmed and the rest of them will be reused. KSM facilitates software reuse, decreasing the effort of developing new applications. Once the complete set of primitives has been programmed, the executable version of the knowledge model is built by duplicating, adapting and assembly primitives using the KSM facilities.

4.3. Characteristics of the KSM Software Environment

The KSM environment helps developers and end-users to construct and maintain large and complex applications, using both knowledge-based and conventional techniques. KSM covers different steps of the life-cycle of an application:

- *Analysis*. KSM uses a particular modeling paradigm, based on the knowledge area concept, for a high level description of the knowledge of the application. The developer uses this paradigm to create a conceptual model to be accepted by the end-user before starting the implementation. Unlike the conventional models of software engineering based on a perspective of information processing, this model is focused on knowledge components which provides a richer and more intuitive description of the architecture of the application. During the analysis phase, the developer follows several steps (the actual realization of these steps may include loops):

    1. *Identification of top-level tasks:* Initially, the developer defines a conversation model between the user and system, where the top-level task are established. This conversation model can be validated with the end-user by developing a mock-up prototype.

    2. *Top-down task decomposition*: Each top-level task is refined by turn selecting its appropriate method (or methods) which decomposes the task into subtasks. This top-down decomposition continues several levels until primary methods are identified. As a result, a set of task-method-subtask-domain hierarchies is produced, one for each top-level task.

3. *Knowledge area integration*: Finally, the components of the task-method-subtask-domain hierarchies are encapsulated in a structure of knowledge areas. Here, different structuring options may be considered until an acceptable one is obtained representing adequately the epxert intuitions.

It is important to note that the analysis phase may be either (1) *totally creative*, i.e. the model is only derived from the information provided by domain experts, or (2) *model-based*, i.e., the model is also derived from a generic model taken from a library of reusable models that establishes the abstract structure of components and relations.

- *Design and implementation*. KSM assists the developer to create the final executable version of the knowledge model. In order to do so, KSM manages reusable software components (called primitives of representation) which are adapted and assembled by the developer following the structure of the conceptual model. Normally, primitives provide general inference procedures and representation techniques to write knowledge bases (although also domain dependent primitivess can be considered). In this phase it is also required to fill in the architecture with the specificities of the problems to be solved. For this purpose the domain models are to be formulated by introducing parameter values and knowledge bases required for case modeling.

- *Operation and maintenance*. Once the application is built, the end user can apply KSM to consult the structure of the conceptual model of the application and may access to local independent knowledge bases following this structure. The role of KSM here is to allow the end-user to open the application to access to its knowledge structure so that, instead of being a black box like the conventional systems, the final application shows high level

comprehensible descriptions of its knowledge. The user also may change the conceptual model at this level, without programming, in order to adapt the system to new requirements. KSM automatically translates these changes into the implementation level.

As it was described, KSM conceives the final application as a modular architecture made of a structured collection of building blocks. At the implementation level, each elementary block is a reusable software module programmed with an appropriate language and a particular technique (knowledge-based or conventional). Using KSM, a developer can duplicate, adapt and assemble the different software components following a high level knowledge model which offers a global view of the architecture. The direct advantages of the use of KSM are: (1) it is easier to design and to develop large and complex knowledge based systems with different symbolic representations, (2) the final application is open to be accessed by the end-user in a structured way, (3) the modular nature of the architecture allows the system to be more flexible to accept changes, and it is also useful for production planning (i.e. it is possible to define an implementation plan according to the structural constraints of the model), and for budgeting (the project is decomposed in understandable components where it is possible to make better prediction of time and costs). The KSM software environment provides the following facilities:

a) A *user interface for knowledge modeling*, following the knowledge area paradigm. This interface consists of: (1) a graphical window-based view of knowledge modules providing visual facilities to create, modify and delete components, (2) the Link language interpreter which allows the developer to formulate high level problem-solving strategies that integrate basic components, and (3) the Concel language compiler to define common

terminologies shared by different modules. Figure 11 presents a general screen presented by the KSM environment showing the knowledge areas components of a structured model.

b) A *library of reusable software components* (the primitives of representation). They may be either conventional or knowledge-based modules. Examples of general knowledge-based primitives are: rule-based primitive with forward and backward chaining inference procedures, frame-based primitive with pattern-matching procedures, constraint-based primitive with satisfaction procedures, etc. The library is open to include new components according to the needs of new applications and they can be programmed by using different languages (C++, Prolog, etc.).

c) A *user interface for execution*. This interface allows the developer to execute knowledge models to validate them. The evaluation may be done either for the whole model or parts of it. Using the interface, the developer may select tasks to be executed, provide input data and consult results and explanations. The execution makes use of the Link interpreter to execute methods and the primitives of representation to execute the basic inferences.

The original version of KSM operated on Unix environments with a minimum of 32 Mb of RAM and more than 50 MIPS of CPU. This version of KSM was implemented using C and Prolog languages. Both languages were improved by adding object oriented features. X Window and OSF/Motif were used to develop the user interface. Recently, a new version for the Windows operating system was constructed using C++ and Java languages.

5. EXAMPLE OF KNOWLEDGE MODEL USING KSM

In order to illustrate the previous knowledge modeling process, an example about traffic management is presented in this section (another detailed example in this domain can be found at Molina et al. (1998b)). Traffic management systems must be reactive to the different states of traffic flow in a controlled network (a network equipped by sensors and data communication facilities allowing to get real time data in a central computer and to diffuse signals from this central computer). These systems evolved from an initial approach based on a library of signal plans which were applied on a time based pattern, to an intelligence for understanding traffic situations in real time integrating a model for decision making (Bretherton et al. (1990), Mauro (1989)). Nevertheless, the experience in using such systems showed deficiencies when the traffic situation became specially problematic and the intervention of the operator was necessary and almost customary in most installations.

The above considerations suggested a need to complement the existing traffic control systems (including pre-calculated plan systems, dynamic systems and VMS systems) with an additional layer where the strategic knowledge, currently applied by human operators, may be applied to understand the specific processes of congestion development, and corresponding actions for alleviating the problem may be modeled. From this viewpoint, the technology of knowledge-based systems was considered adequate for designing and implementing suitable knowledge structures to formulate conceptual models for traffic analysis and management.

To control a motorway a technique usually applied is to send messages to the drivers through panels whose content can be modified by operators at the control center (these panels are named Variable Message Signals (VMS)). The motorway is adequately controlled if in any

moment the message panels are pertinent and consistent with the state of the traffic flows in the motorway. An intelligent system to help in decision support could be formulated in terms of a general *propose and revise* method where the current state of the panel messages are evaluated with respect to the state of traffic and are revised accordingly. The following analysis is performed to design such simple system.

5.1 The Domain Knowledge

The basic traffic vocabulary of this example includes the following concepts (figure 12). There are traffic detectors with three attributes, occupancy, speed and flow that get as value temporal series of numerical values. A road section is a significant cross point of the road that is characterized by its capacity (maximum flow that the section accepts), the detector associated to the road section, and two dynamic qualitative attributes, saturation level and circulation regime, that are useful to characterize the current state of the section with symbolic values. The road link serves to connect consecutive sections. It is characterized by the upstream and the downstream section. Thus, the structure of the road network is represented by a set of road links. A path is a sequence of links and includes the attribute travel time. Finally, there are variable message signs (VMS) where it is possible to write messages for drivers. Two types of meaningful messages can be considered in this simplified example: (i) qualification of the traffic state downstream the panel location (e.g., "slow traffic ahead at N Km.", "congested link at N Km") and (ii) information on time delays to reach some destination (e.g., "to destination D, 20 min by option B").

The declarative knowledge of the domain model is based, on the one hand, on conditions relating traffic states in some sections and the possibility of writing a message in a particular VMS. Each panel includes a predefined set of messages (messages qualifying the situation in a set of traffic sections and messages for path recommendation) where each message should be presented to the drivers when certain conditions about the situation in several downstream sections are satisfied. On the other hand, there are also conditions modeling consistency between messages along a path or in an intersection to ensure that the drivers along a path do not find contradictory recommendations or that the drivers incoming a roundabout are adequately directed by the messages to select the adequate options. In order to represent the first type of knowledge, rules can be written using the following format:

```
if    circulation regime of section Si = Ri,
      saturation level of section Si = Li,
      circulation regime of section Sj = Rj,
      saturation level of section Sj = Lj
      ...
      travel time of path Pn = close to N minutes,
      travel time of path Pm = close to M minutes,
      ...
then  message of panel Mk = Tk
```

This type of rules model the fact that when the sections *Si*, *Sj*, ... (sections that are downstream the panel *Mk*) are in a certain state characterized by the circulation regime and the saturation level of such sections, and when the estimated travel time of certain paths are close to N minutes (where the evaluation of the *close to* qualifier can be done by using a particular fuzzy possibility function), then it is deduced that the panel *Mk* should present the

message *Tk*. Thus, each panel will have a set of this type of rules that establish conditions about the sections downstream the panel for each type of message to be presented.

The second type of conditions that establish consistency between messages may be modeled by two classes of rules. First, a set of rules to deduce sets of messages for panels based on messages of other panels, with the following format:

```
if     message of panel Mi = {Ti,...},
       message of panel Mj = {Tj,...},
       ...
then   message of panel Mk = {Tk,...},
       message of panel Ml = {Tl,...},
       ...
```

This type of rule means that the occurrence of messages $\{T_i, \ldots\}$ for panel M_i, and messages $\{T_j, \ldots\}$ for panel M_j implies that the messages of panel M_k and panel M_l must be respectively included in the sets of messages $\{T_k, \ldots\}$ and $\{T_l, \ldots\}$. The previous rules are complemented with rules defining no-good representing incompatible sets of messages. This type of rules present the following format (meaning that is not possible the occurrence of messages $\{T_i, \ldots\}$ for panel M_i, and messages $\{T_j, \ldots\}$ for panel M_j):

```
if     message of panel Mi = {Ti,...},
       message of panel Mj = {Tj,...},
       ...
then   no-good.
```

In addition to the previous declarative model, there are also abstraction methods to determine (1) the qualitative values of road sections for qualification of the saturation level and circulation regime, based on the numerical values recorded by detectors for occupancy, flow and speed, and (2) the estimated travel times for the predefined paths, based on the current speed registered on detectors.

5. 2. The Strategy of Inference

The standard general reasoning method usually applied in this type of control applications is based in three main steps: (1) *problem detection*, where possible situations with significant differences with respect to the features of a goal situation are identified (in the case of traffic this ideal situation is the flow on the road with no congested areas) (2) *problem diagnosis*, when an undesirable situation is presented, its potential causes are identified, and (3) *problem repair*, an analysis of the possible sets of actions capable to modify the causes in positive terms is performed to select the adequate ones. This is the approach followed in the KITS project Boero et al. (1993, 1994) and the TRYS system Cuena et al. (1995, 1996a, 1996b). Although this is a right approach it is also possible to use as alternative a shallow model where a direct relation between the state of traffic and the panel messages is established as the one proposed in this model, where the diagnosis and repair steps are summarized in a single situation-action relationship, Cuena (1997).

The inference procedure follows a general strategy based on a cycle of reasoning that *updates* the set of messages that were determined in the previous cycle, according to the current state of the road network. This updating is a kind of reconfiguration task that first *abstracts* the

current state using information recorded by detectors, then *removes* not valid messages (messages whose conditions to be presented are not satisfied), and finally *extend* the current set of messages with new ones. This strategy can be modelled by a task-method-domain structure (figure 13) where the global task called *determine messages configuration* is divided into three subtasks. The first task abstracts information from detectors using a network of abstraction functions to determine qualitative values (e.g., for control regime and saturation level) and also uses elementary numerical procedures in order to compute derived numerical values (such as the estimated travel time and the exact numerical value for saturation). The second task, *remove not valid messages,* studies the pertinence of each message by checking their applicability conditions. Finally, the third task, *extend with new messages*, is applied for the panels that have removed their messages. It can be carried out by a propose-and-revise method with three subtasks: propose new message, verify messages consistency and remedy violations. The first subtask, *propose new messages*, may be performed by a rule based forward chaining method which uses a knowledge base of rules proposing possible messages in panels. These rules present the format of the first type of rules presented in the previous section. The next subtask, verify message consistency, may be performed by a method using no-good rules (of the second type of rules presented in the previous section) for the definition of contradictory messages along paths in the network. Finally, the remedy violations subtask solves inconsistencies by retracting inadequate messages. This subtask may be performed using a base of priority rules providing criteria for selection of messages candidates to retract after the results of the two previous subtasks. The last subtask applies a minimum retraction criterion, i.e. it is selected the minimum set of messages according to the priority scheme that ensure the consistency. These three tasks work in a loop using the following control mechanism. First, a new message is proposed for a single panel that does not have a message yet. Then the global consistency is studied. If a violation is found, then the remedy task

decides which panel must change its message and a new proposal is generated, starting a new cycle. This process finishes successfully when all panels have at least one message and they do not present inconsistencies.

The previous analysis may be summarized in terms of the knowledge-area structure presented in figure 14. This structure includes a top-level area representing the whole model, called VMS management knowledge, that includes the task called *determine messages configuration*. This area is decomposed into two simpler areas: *abstraction knowledge*, that includes the criteria to abstract data from detectors, and *messages knowledge*, that represents the knowledge about messages of panels. This second area is decomposed into three: *messages applicability, messages consistency* and *preference criteria*. Note that each primary area (bottom-level area) encapsulates one type of domain model defined in the previous task-method-domain structure which corresponds to a type of knowledge base (KB), together with the associated primary tasks. For instance, the domain model called applicability conditions for messages is part of the primary area called messages applicability and also includes the two tasks that make use of this model: *remove not valid messages and propose new messages.* In this case the message applicability is evaluated using fuzzy possibility distributions of every message defined with respect traffic flow, speed. A message is discarded when its possibilities with respect to the current and predictable traffic conditions are unacceptable.

5.3 Generalization to Support Reuse

A more realistic and complex model may be designed if a larger area for traffic management is considered. In this case, an appropriate approach is to divide the whole traffic network into

local regions in such a way that first, a decision about messages for panels is locally carried out taking into account the specific problems of each region and, then, the local proposals are combined avoiding incompatibilities (given that there may be common panels to several regions). The previous model can be used to locally propose messages for the region panels and it needs to be extended with other knowledge areas that include knowledge to manage the whole network. Figure 15 shows the complete structure of knowledge areas where additionally to the knowledge areas of the previous model, a new top-level area has been included that contains the model for a region together with another new area responsible for combining local proposals. The combination is based on a kind of generate-and-test strategy where first a combination is generated based on the local proposals and, then, a test is performed to detect conflicts produced by inconsistent proposals in the panels common to two areas. As an alternative to this design, the proposed architecture using region control models could be also considered like a multiagent system, where the interaction between regions could be modelled by social domain model to solve cooperation and consensus formation in dealing with common problems. Some experiments in this direction can be found at Ossowski et al. (1996).

Note that the model that figure 15 presents is a generic structure considered as a pattern to be instantiated using the particular knowledge of a specific traffic network. In doing so, there will be several instances of the knowledge structure for a region, all of them having the same set of classes of knowledge areas and the same tasks and methods, but with different knowledge bases. Therefore this design presents already a certain level of generality to be reused for different traffic networks of different cities. However, in order to increase the level of reuse it is interesting, when it is possible, to abstract its components and to propose a more general knowledge structure, capable to be used for other domains. The appropriate level of

abstraction is a decision that the developer must establish according to the possibilities of each model and taking into account that too general methods may be difficult to be used by other developers.

In this example, the previous described model can be abstracted in order to build a more general version, independent of the traffic domain. Figure 16 shows the task-method-domain structure corresponding to the structure that figure 13 presents, that has been extended to consider several components (regions in the traffic problem) and tasks, methods and domain models have been abstracted from the traffic domain. Here, the set of messages for panels are generalized to be considered as a set of values for parameters, and the selection of the traffic control plan is viewed as a configuration problem, i.e., updating values to parameters according to certain constraints.

Likewise, figure 17 shows the generalized knowledge-area structure corresponding to the structure presented in figure 15. This structure includes traffic-independent tasks and knowledge areas. The operationalization of this model using KSM requires first (1) to write the corresponding Link methods for non primary tasks, (2) to write conceptual vocabularies using Concel and (3) to select appropriate primitives of representation that implement primary areas. In this case, for instance, the knowledge base for applicability conditions can be written using rule-based representation, and the combination constraints base can use a constraint-based representation. Thus, primitives that follow these representations and implement the corresponding inference procedures can be used here as reusable software components to support this part of the model. Finally, once this generic architecture has been built, it can be used as a pattern to construct a particular domain model by creating instances of the generic knowledge areas and writing specific vocabularies and knowledge bases .

6. DISCUSSION

The proposed approach may be evaluated according to the potential capacities and drawbacks to support applications.

6.1. The Capacities

- *Structuring & encapsulation support*: it is produced through the knowledge areas structure with the advantage that the modularity of structuring is based in a conceptual organization close to the common sense understanding of the model. An example of structure is summarized in figure 15, where the relations between knowledge areas are *is part of* type and the internal structure of the areas in terms of tasks and knowledge components may be inspected. This structure allows the user to understand and to maintain in acceptable conditions a given application. The usual structuring principle of application in software engineering is based on the data and process components. These concepts are less close to the common sense intuitions of users non computer literate so the new organization based on knowledge level structuring is a more adequate solution for encapsulation and structure. Moreover, this organization may be implemented in object oriented software models which means that the advantages of this type of implementation are implicit in the approach based on knowledge areas.

- *Software sharing support*: The components of a generic model may be replicated several times with different contents in a case model. In fact, the proposed concept of knowledge area based encapsulation allows a new level of abstraction of applications as it is displayed

in the figure 18 where they are presented the three abstraction levels based provided by the KSM environment where:

- In the lower level a collection of procedures to interpret the basic knowledge representation entities is included together with the software for editing and maintaining an application.

- In the following level there is a generic model abstraction with a generic structure of knowledge areas and where all the composed inference methods are formulated.

- The generic methods of the previous level use different domain models as presented in the case model level.

• *Reusability*: Two types of reuse may be possible:

- Reuse of the basic units representing primary methods of reasoning supported by rules, frames, constraints, tables etc. The reuse is produced at the level of the problem solving methods where different knowledge bases of constraints, rules, etc. are formulated for every case.

- Reuse of total or partial generic models. In this case reuse is produced by importing an upper level reasoning method of a generic model that may be used to perform a subtask in other model.

Reusable software components are self contained, clearly identifiable artefacts that describe and/or perform specific functions, have clear interfaces, appropriate documentation and a defined reuse status Sametinger (1997). The reuse status contain information about who is the owner of a component and who maintains it. Every component, then, must include a description of its functionality and a code which performs it. The problem is that as a consequence of the maintenance process it may happen that some discrepancies between the textual description of the component and the software code exist. This may be the source of misunderstandings in the use and performance of the component. Obviously, a knowledge area (including the lower level units supporting its functionality) is a typical software component which has the advantage that its code written in knowledge representation language is understandable by the users even not expert in Computer Science. Then, it is not required to include the double aspect of textual description and software code to get an understandable formulation of a software component. This is an advantage for maintenance and applicability of the components formulated in a knowledge based approach.

- *Support of the software production process* through the generic model abstraction. This is justified by: (1) generic models is a form of specification of an application class contents, (2) possibility to schedule and to budget case applications of a generic model because all the elements to be developed are well defined in this type of models where it is explicit a structured organization of components every one well known and, hence, easy to evaluate in time and cost, and (3) certain level of maintenance by users.

- *Advanced human-computer interaction support*: The usual approach to human computer interaction is to include in the application facilities to generate versions of the system

answers closer to the user needs and understanding facilities (natural language expression, multimedia presentation of answers, etc.). However, not too much attention has been paid to allow the user to enter in the conceptual world of an application because, usually, it is implemented as a fixed set of functions supported by a blackbox implementation using conventional languages not understandable enough by the users. Using knowledge based models it is possible a type of interface where for every class of questions a task-method-structure is designed supported by declarative domain models allowing the production of explanations at different levels and, hence, to allow the user to communicate at the knowledge level with the application.

6.2 The Drawbacks

There are two types of critiques for the knowledge modeling approach:

- *Knowledge building & validation process*: To ensure that the right contents are introduced in a case model, an experimental approach must be applied. The current state of knowledge acquisition methods ensure more reliable (and predictable in terms of time and budge) development processes. As commented before using structured knowledge representation

models it is possible to identify in the phase of specification a structured view of the knowledge of an application in terms of hierarchical structures of problem solving methods and associated domain models. This generates a framework to guide the implementation of the reasoning method and to the elicitation of domain models in terms of adequate size which will ensure reliability and efficiency. Moreover, if qualified operators are used to develop applications and advanced human-computer interaction environment is used, as it is possible now, the knowledge debugging will be efficient enough because it is possible to obtain explanations at different depths so it will be possible to establish the role of the different levels of knowledge used to produce and answer.

- *Efficiency*: Although the knowledge based models work in an interpreted mode, which is a drawback for efficiency, the hardware evolution already shows power enough to ensure an acceptable timing in the answers even in these conditions.

7. CONCLUSIONS

An outline of the possible profile of tools for application development based on knowledge modeling technologies has been proposed using as example the approach supported by the KSM tool. Although alternative approaches may be used to improve some of the specific drawbacks that the KSM tool may present, our experience shows that it is possible to conceive software development platforms supporting something that could be named *cognitive programming* where data and procedures specification are included as pieces of

knowledge. At the current state of this technology, it may provide, both from the point of view of operation and development, levels of service competitive enough with the conventional approaches because, as commented in the previous paragraph, it is possible to satisfy most of the conditions usually asked for a good software development platform and methodology with additional advantages not usually considered in conventional approaches that are being focus of growing attention such as potentialities for reuse and advanced user interaction support.

8. ACKNOWLEDGEMENTS

B. Chandrasekaran provided valuable comments to an earlier version of this paper, the PhD and undergraduate students of the Intelligent Systems Group (ISYS group) provided help in implementation of KSM. Reyes Riera was the responsible for edition.

9. REFERENCES

BOERO M. & CUENA J. & KIRSCHFINK H. & KROGH C. (1993): "KITS: A General Approach for Knowledge-Based Traffic Control Models" *Proc. Technical Days on Advanced Transport Telematics*, Brussels, March 1993.

BOERO M. & CUENA J. & KIRSCHFINK H. & TRAETTEBERG H. & WILD D. (1994): "The Role of Knowledge-Based Models in Traffic Management and their Design" in *Towards an Intelligent Transport System* Vol. 2, edited by ERTICO. Paris, 1994.

BRETHERTON R.D. & BOWEN G.T. (1990): “Recent Enhancements to SCOOT - SCOOT version 2.4”, *Proc. of the 3rd International Conference on Road Traffic Control, IEE*, London, 1990.

CHANDRASEKARAN B. (1983): “Towards a Taxonomy of Problem Solving Types” *A.I. Magazine* 4 (1) 9-17, 1983.

CHANDRASEKARAN, B. (1986): "Generic Tasks in Knowledge Based Reasoning: High Level Building Blocks for Expert Systems Design" *IEEE Expert*, 1986.

CHANDRASEKARAN B. & JOHNSON T.R. & SMITH J.W. (1992): “Task-Structure Analysis for Knowledge Modeling”. *Communications of the ACM*, vol 35, nº 9. September, 1992. Also in *Knowledge Oriented Software Design* J.Cuena (ed). Elsevier, 1993.

CUENA J. (1997): "Traffic Control As Non Monotonic Reasoning: Truth Maintenance Systems For VMS Panels Messages" in *Proc. 5th European Congress on Intelligent Techniques and Soft Computing (EUFIT'97)*, Vol.1, pp 2005-2008, Elite Foundation, Aachen, 1997.

CUENA J. & MOLINA M. (1994): “KSM: An Environment for Knowledge Oriented Design of Applications Using Structured Knowledge Architectures” in *Applications and Impacts. Information Processing’94*, Volume 2. K. Brunnstein y E. Raubold (eds.) Elsevier Science B.V. (North-Holland), 1994 IFIP.

CUENA J. & HERNÁNDEZ J. & MOLINA M. (1995): "Knowledge-based Models for Adaptive Traffic Management Systems" *Transportation Research Part C,* Issue 3 (5), 1995.

CUENA J. & HERNÁNDEZ J. & MOLINA M. (1996): "Knowledge Oriented Design of an Application for Real Time Traffic Management: The TRYS System" in *European Conference on Artificial Intelligence (ECAI'96).* W.Wahlster (ed.), Wiley, 1996.

CUENA J. & HERNÁNDEZ J. & MOLINA M. (1996): "An Intelligent Model for Road Traffic Management in the Motorway Network around Barcelona" in *Advanced IT Tools.* N.Terashima, E.Altman (eds.). IFIP, Chapman & Hall, 1996.

CUENA J. & HERNÁNDEZ J. (1997): "An Exercise of Knowledge Oriented Design: Architecture for Real Time Decision Support Systems" in *Knowledge-Based Systems-Advanced Concepts, Techniques and Applications*. Spyros G. Tzafestas (ed). World Scientific Publishing Company, 1997.

CUENA J. & MOLINA M. (1997): "KSM: An Environment for Design of Structured Knowledge Models" in *Knowledge-Based Systems-Advanced Concepts, Techniques and Applications*. Spyros G. Tzafestas (ed). World Scientific Publishing Company, 1997.

GRUBER T.R. (1993): "A Translation Approach to Portable Ontology Specifications". *Knowledge Acquisition*, 5, 1993.

LANDES, D. & STUDER R. (1994): "The Design Process in MIKE", *Proc. 8th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW'94* (Banff, Canada, January 30-February 4), 1994.

LOWRY M. & DURAN R. (1989): "Knowledge-based Software Engineering", chapter XX in *Handbook of AI* (Vol IV), Barr A., Cohen P.R. and Feigenbaum E. (eds.), Addison Wesley, 1989.

MAURO V. & DI TARANTO (1989): "UTOPIA" *Proc. 6th IFAC/IFORS Conference on Control, Computers and Communications in Transport.* Paris, 1989.

McDERMOTT J. (1988): "Preliminary Steps Toward a Taxonomy of Problem Solving Methods" in *Automating Knowledge Acquisition for Expert Systems*, S.Marcus ed., Kluwer Academic, Boston, 1988.

McINTYRE A. (1993): "KREST User Manual 2.5" Vrije Universiteit Brussel, AI-Lab. Brussels, 1993.

MOLINA M. & SIERRA J.L. & SERRANO J.M. (1998a): "A Language to Formalize and to Operationalize Problem Solving Strategies of Structured Knowledge Models" *8th Workshop on Knowledge Engineering: Methods and Languages KEML 98*. Karlsruhe, Alemania, 1998.

MOLINA M. & HERNÁNDEZ J. & CUENA J. (1998b): "A Structure of Problem Solving Methods for Real-time Decision Support in Traffic Control" *International Journal of Human and Computer Studies* (1998) 49.

MOLINA M. & SIERRA J., CUENA J. (1999): "Reusable Knowledge-based Components for Building Software applications: A Knowledge Modelling Approach" *International Journal of Software Engineering and Knowledge Engineering*, 1999.

MUSEN M.A. & TU S.W. (1993): "Problem-Solving Models for Generation of Task-Specific Knowledge-Acquisition Tools" in *Knowledge Oriented Software Design*, J.Cuena (ed.), Elsevier, 1993.

NEWELL A. (1982): "The Knowledge Level" in *Artificial Intelligence* Vol 18 pp 87-127.

OSSOWSKI S. & GARCÍA-SERRANO A. & CUENA J. (1986): "Emergent Co-ordination of Flow Control Actions Through Functional Co-operation of Social Agents". *European Conference on Artificial Intelligence ECAI'96*, W.Wahlster (ed.), Wiley, 1996.

PRESSMAN R.G. (1992): "Software Engineering: A Practitioner's Approach", 3ª edition, McGraw-Hill, 1992.

SAMETINGER J. (1997): "Software Engineering with Reusable Components" Springer Verlag, 1997.

SMITH D. (1988): "A Knowledge based Software Development System", *Proc. AAAI-88 Workshop on Automating Software Design.* St. Paul, Minnessotta, 1988.

STEELS, L. (1990): "Components of Expertise" *AI Magazine*, Vol. 11(2) 29-49.

WIELINGA B.J. & SCHREIBER A.T. & BREUKER J.A. (1992): "KADS a modeling approach to knowledge engineering " in *Knowledge Acquisition* 4, 1992.

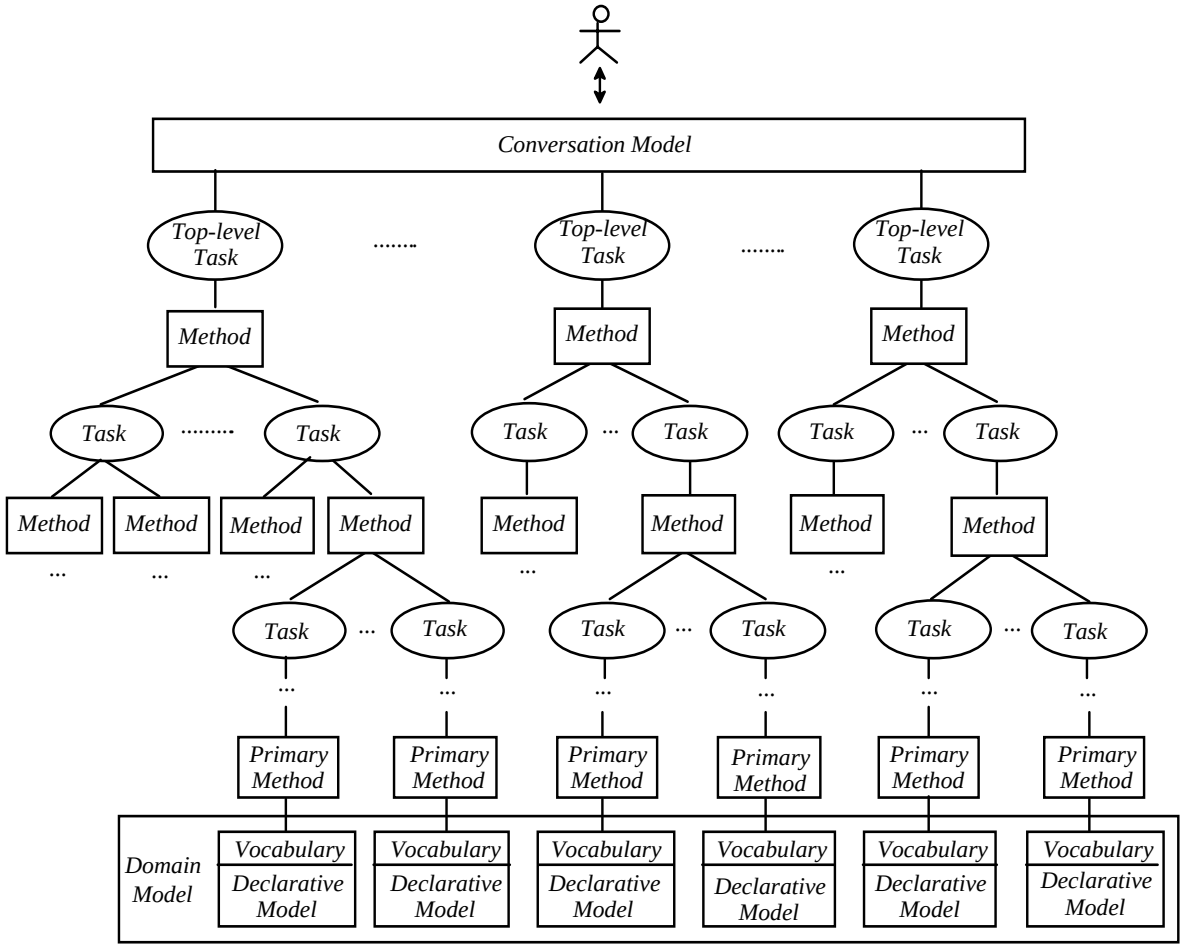

Figure 1: Task-method-domain structure for a model

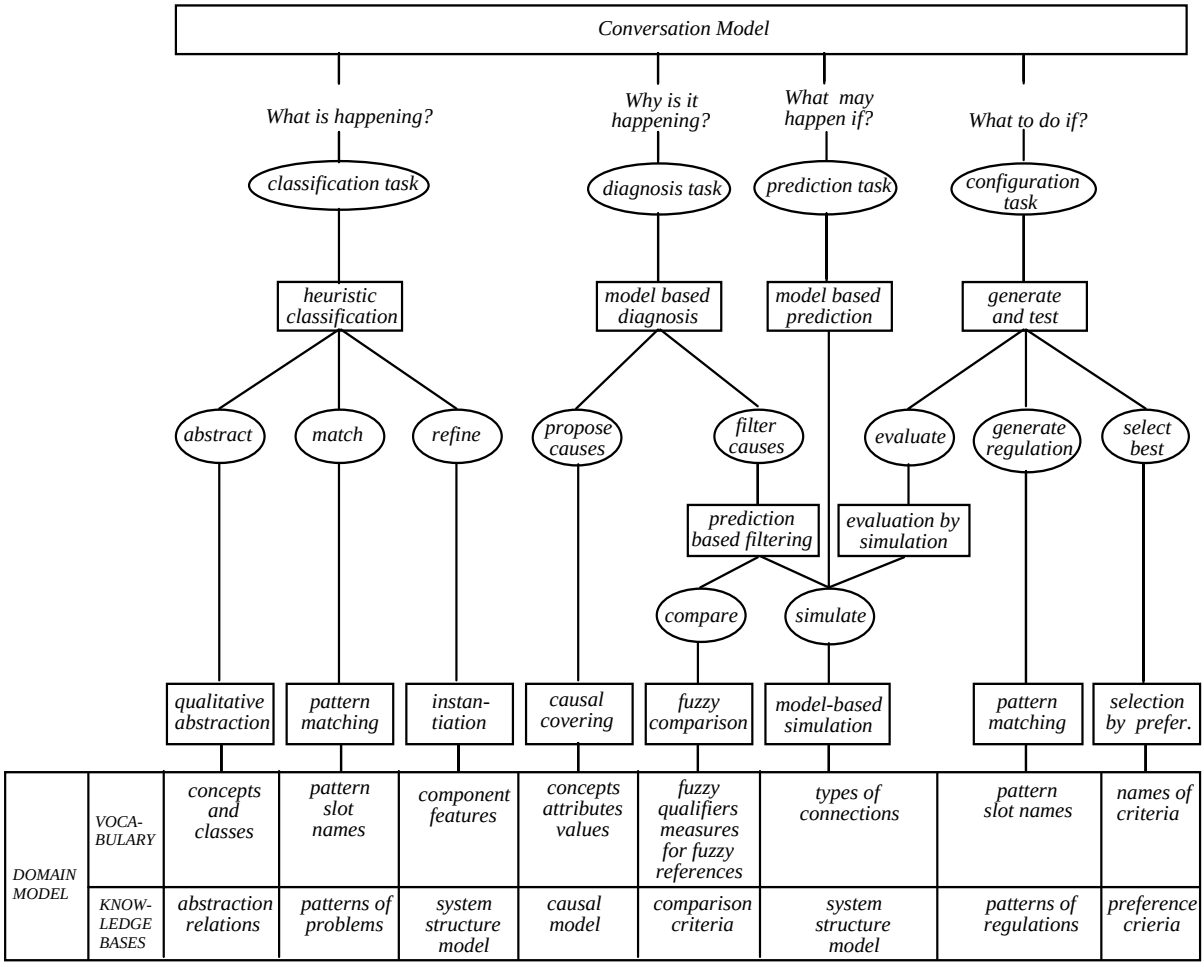

Figure 2: Example of task-method-domain structure

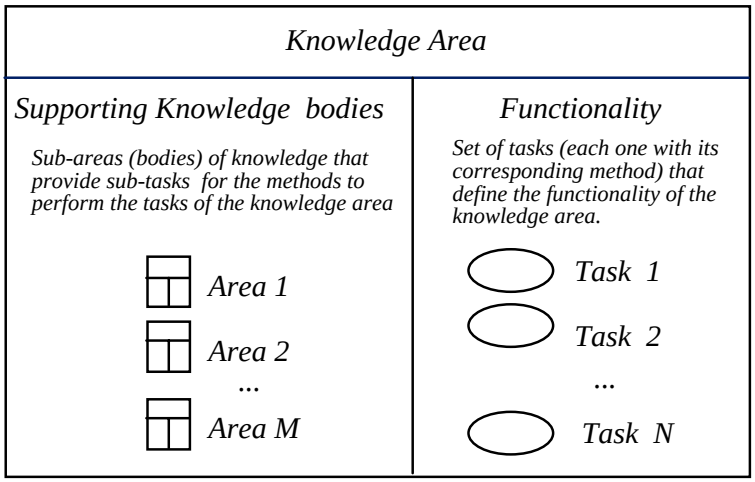

Figure 3: Format of a knowledge area

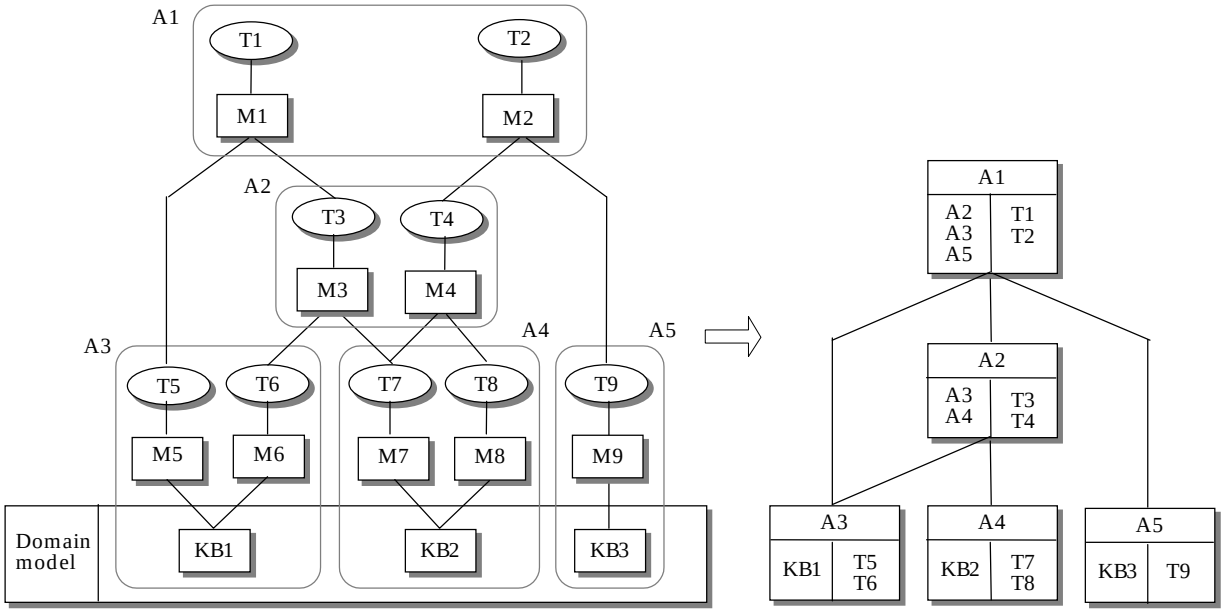

Figure 4: Abstract example of the encapsulation provided by the knowledge area concept. The hierarchy of task-method-domain on the left can be grouped using five knowledge areas A1, A2, ..., A5. This produces the knowledge-area view on the right which offers a more synthetic view of the model

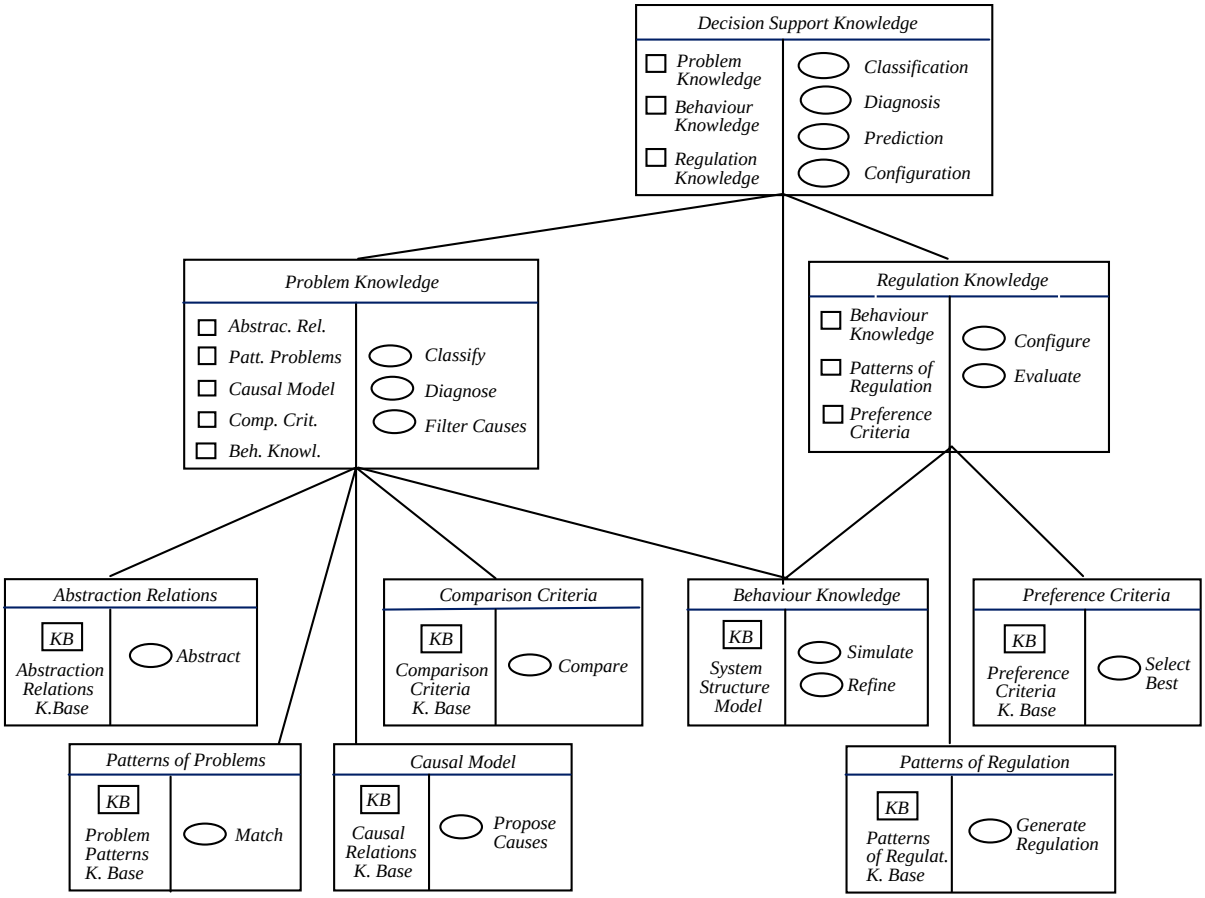

Figure 5: Knowledge-area view of the generic decision support model

```
CONCEPT name-of-concept {SUBCLASS OF | IS A} name-of-class.
[ATTRIBUTES:
      name-of-attribute facet facet ... facet,
      name-of-attribute facet facet ... facet,
      ...
      name-of-attribute facet facet ... facet.]
```

Figure 6: General format of the formulation of a concept using the Concel language.

```
METHOD method-name

 ARGUMENTS
        [INPUT list-of-inputs]
        [OUTPUT list-of-outputs]

 DATA FLOW
        data-connection-among-subtasks

 [CONTROL TASKS
        data-connection-among-control-tasks]

 CONTROL FLOW
        rules-to-determine-the-control-regime

 [PARAMETERS
        default-values-for-parameters]
```

Figure 7: General format of a method formulation using the Link language.

```
(knowledge-area-name) task-identifier
[INPUT
          { [mode] flow-expression } + ]
[OUTPUT
          [mode]  { flow-identifier } + ]
```

Figure 8: General format of an input/output specification using the Link language.

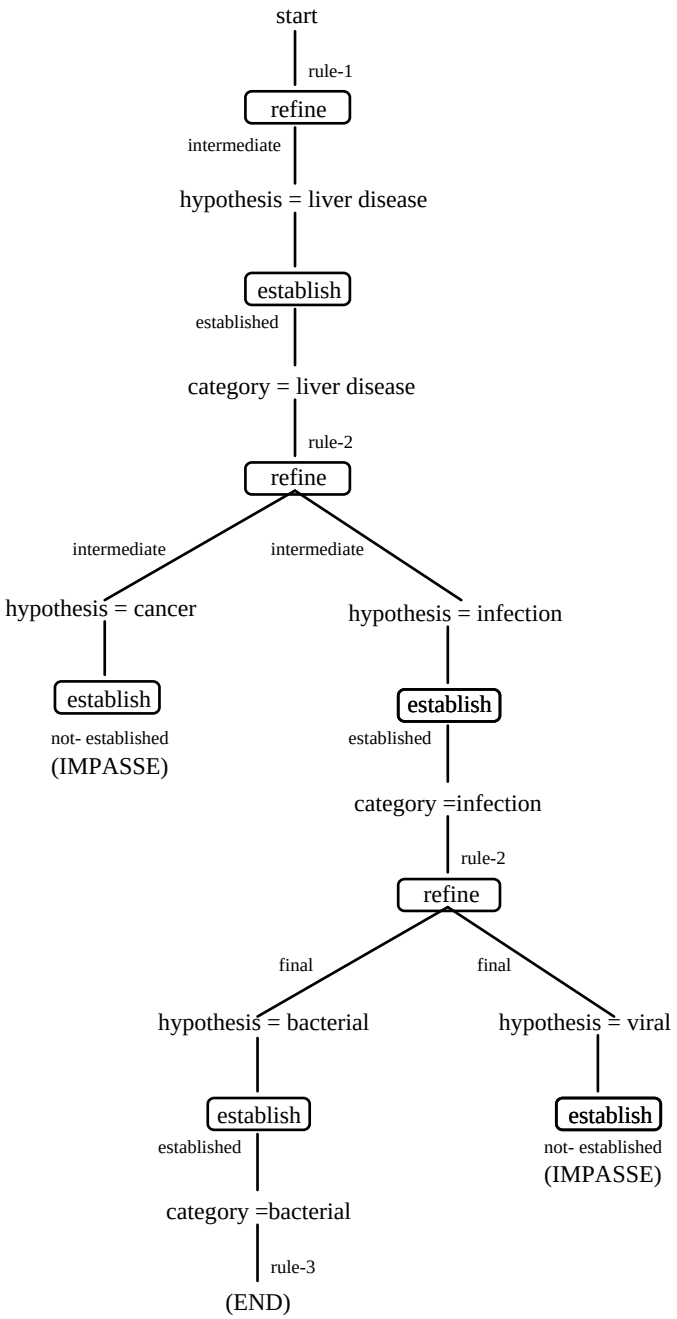

*Figure 9: Example of the execution of a method where there is a non linear reasoning (example taken f*rom Chandrasekaran et al. *(1992) and adapted to the Link operation).*

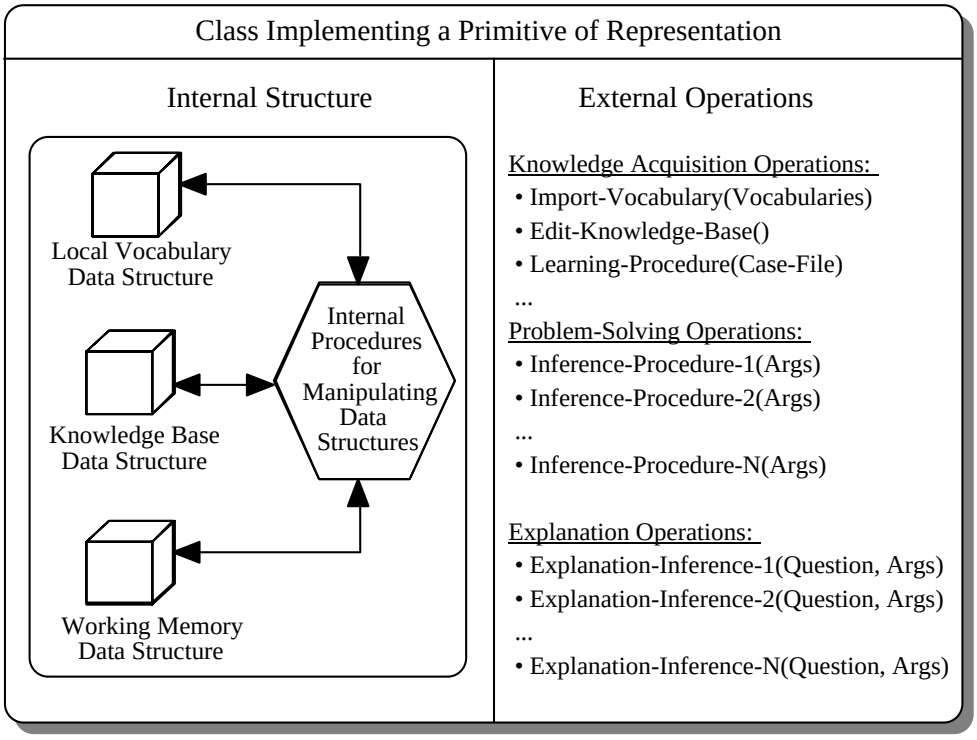

Figure 10: Structure of the class implementing a primitive of representation

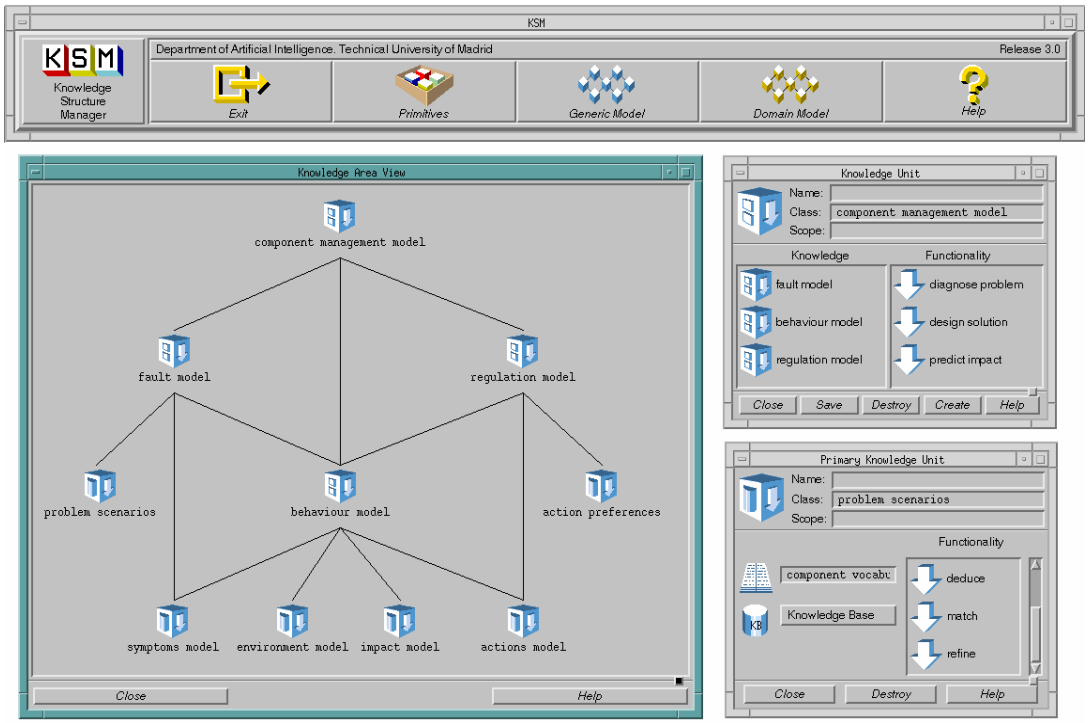

Figure 11: Screen example of the KSM environment.

Clases of concepts	*Attributes*	*Values*
Di: Detector	*occupancy* *speed* *flow*	*temporal series of num.values in %* *temporal series of num.values in Km/h* *temporal series of num.values in veh/h*
Si: Road Section	*capacity* *saturation level* *circulation regime* *detector*	*number in veh/h* *{free, critical}* *{fluid, unstable, congested}* *instance of detector*
Li: Road Link	*upstream section* *downstream section*	*instance of section* *instance of section*
Pi: Path	*links* *travel time*	*list of instance of links* *number in minutes*
Mi: Variable Message Sign	*message*	*{"slow traffic ahead at N Km",* *"congested link at N Km",* *"to area A, 20 min by option B",* *...}*

Figure 12: Basic vocabulary used in the traffic domain model

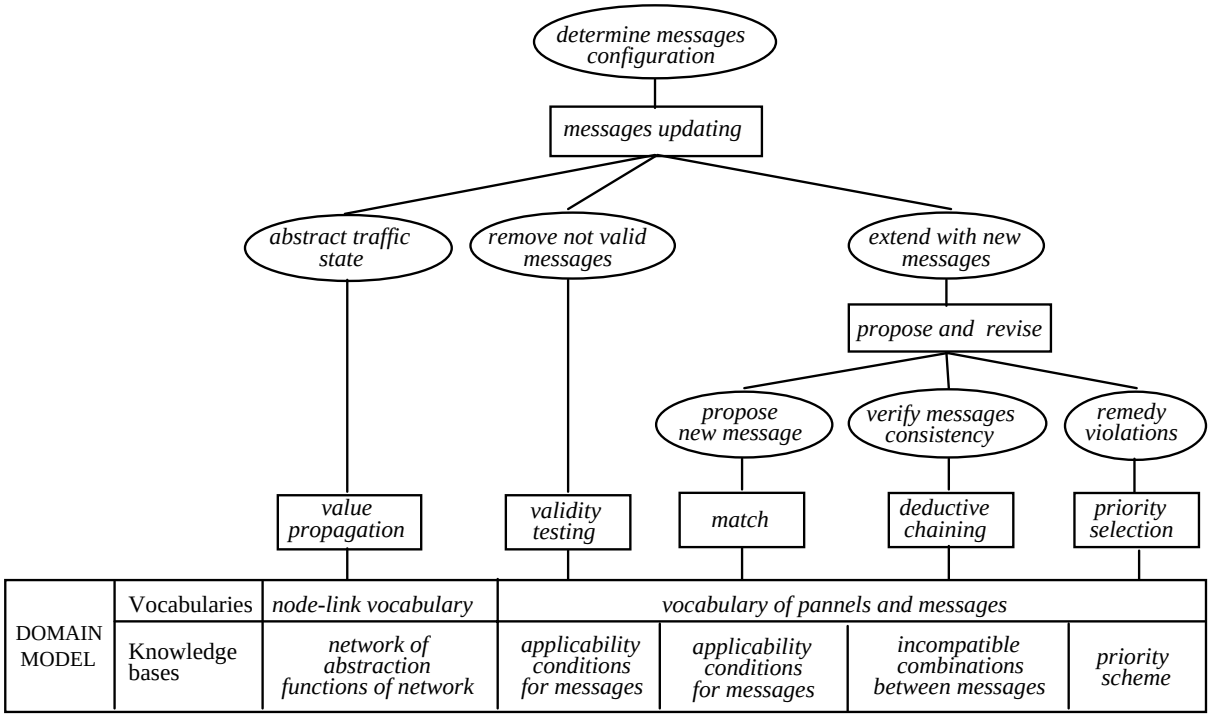

Figure 13: Task-method-subtask-domain structure for the traffic control example

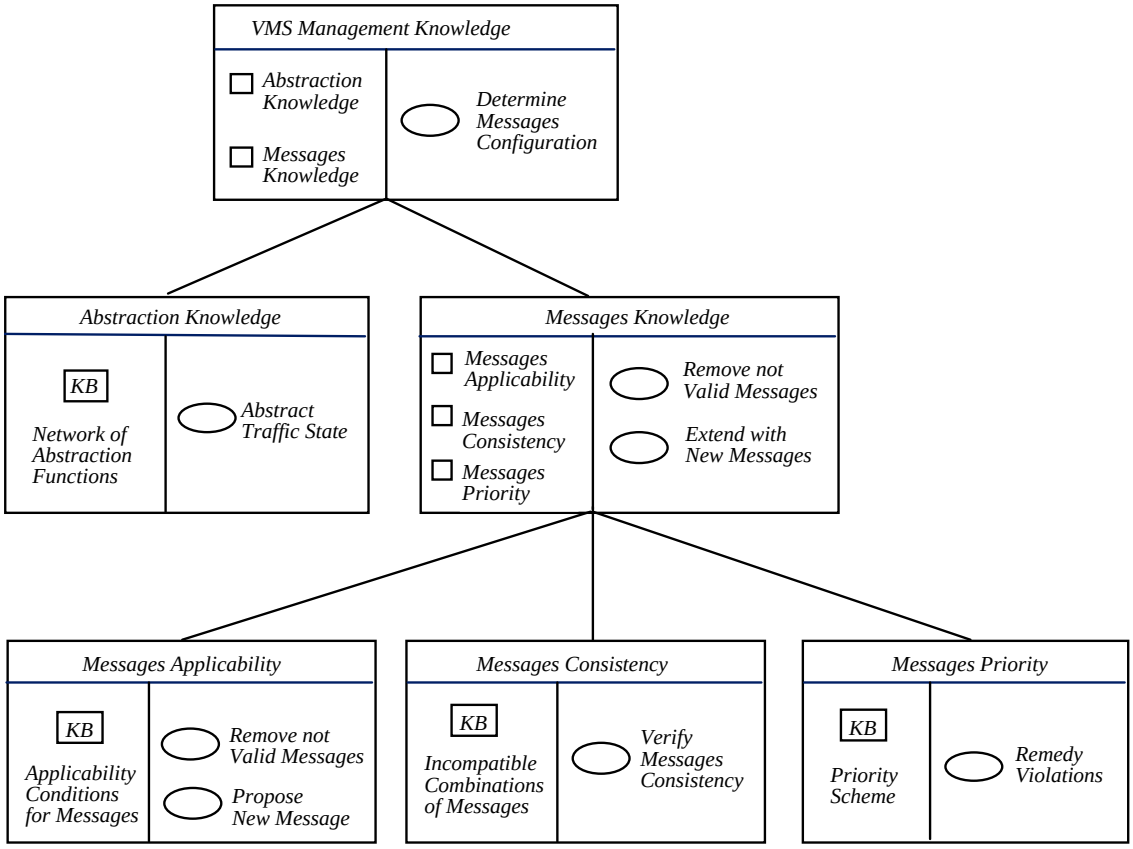

Figure 14: The knowledge-area view of the knowledge model for road VMS management

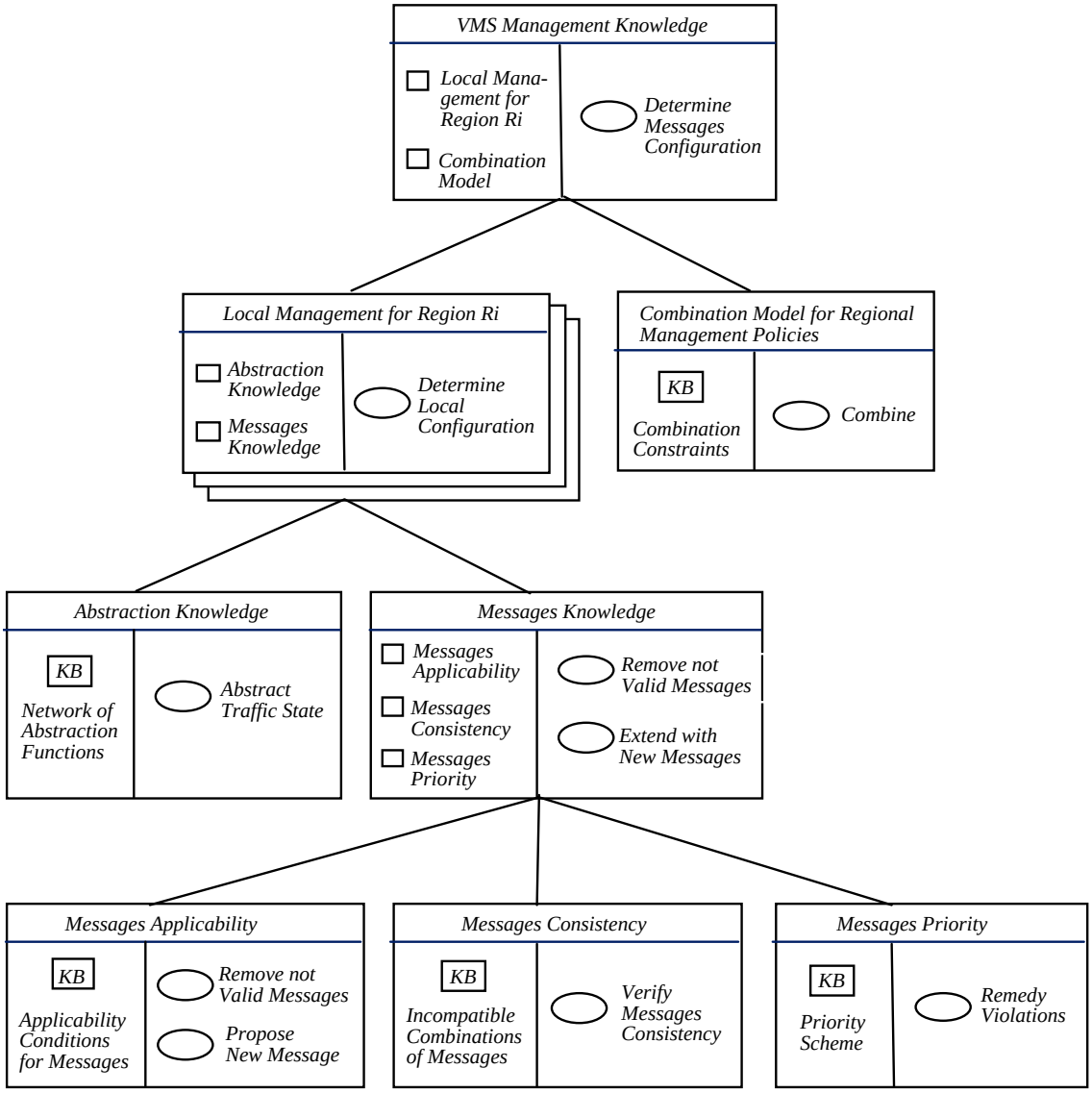

Figure 15: Knowledge-area structure corresponding to an extension of the model of figure 13 where a more complex traffic network has been considered that needs to decompose the total network into simpler regions.

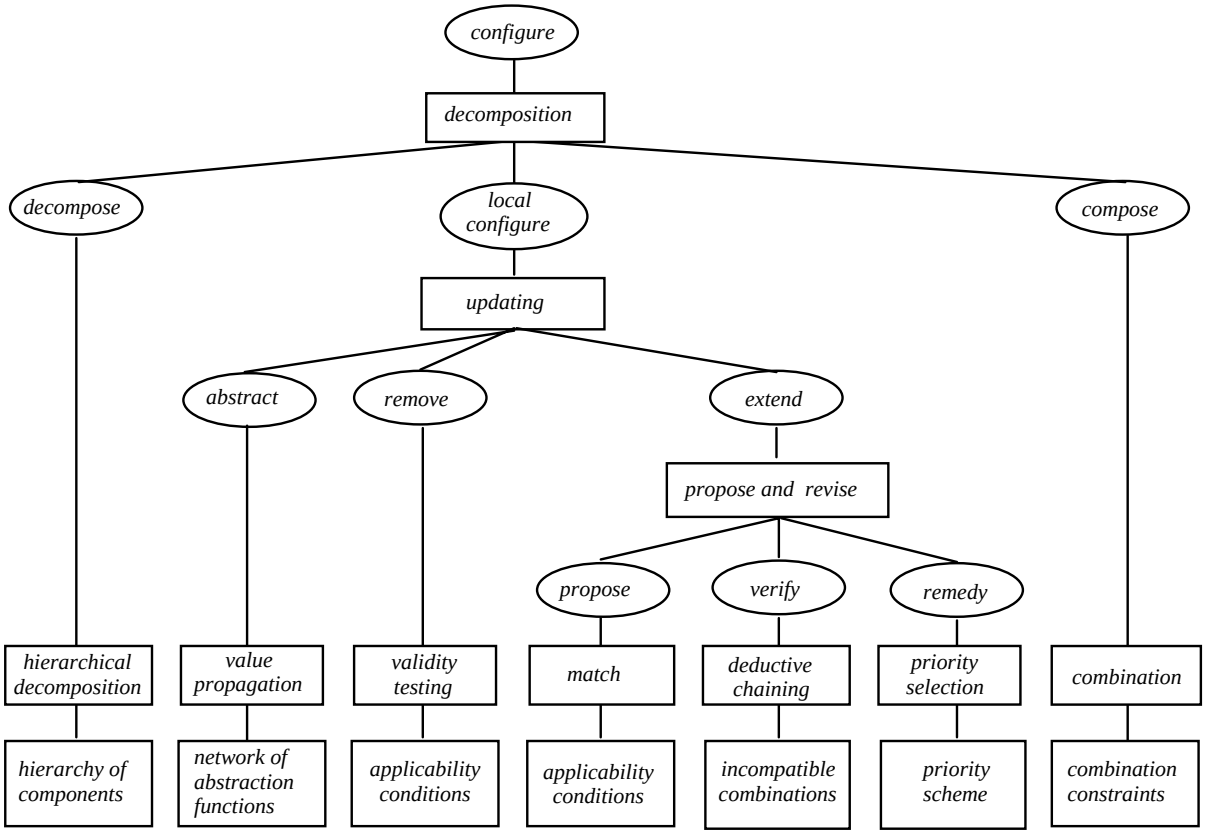

Figure 16: Task-method-domain structure corresponding to an extension and abstraction of the structure established for the traffic problem (shown in figure 12).

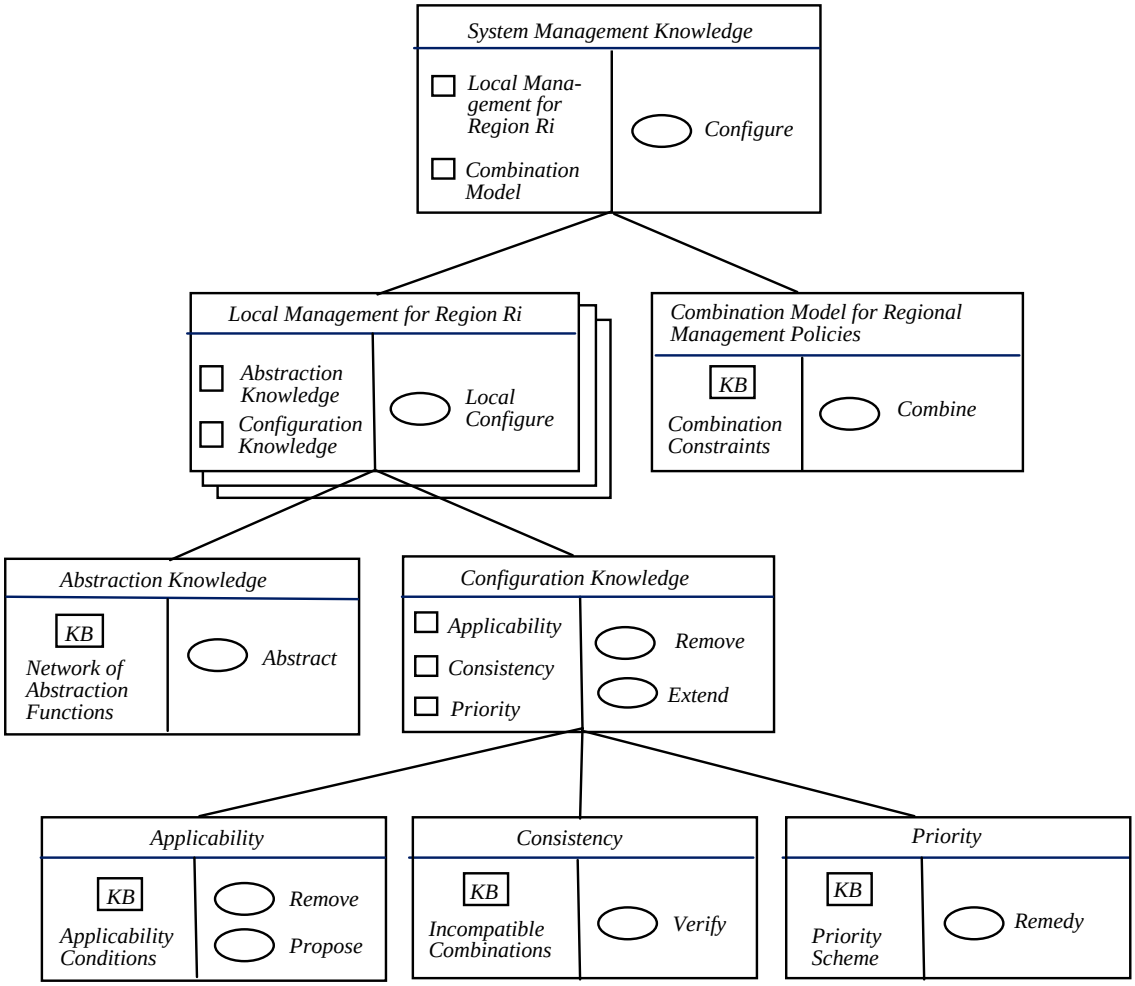

Figure 17: Knowledge-area structure corresponding to an abstraction of the structure designed for the traffic problem (figure 14) .

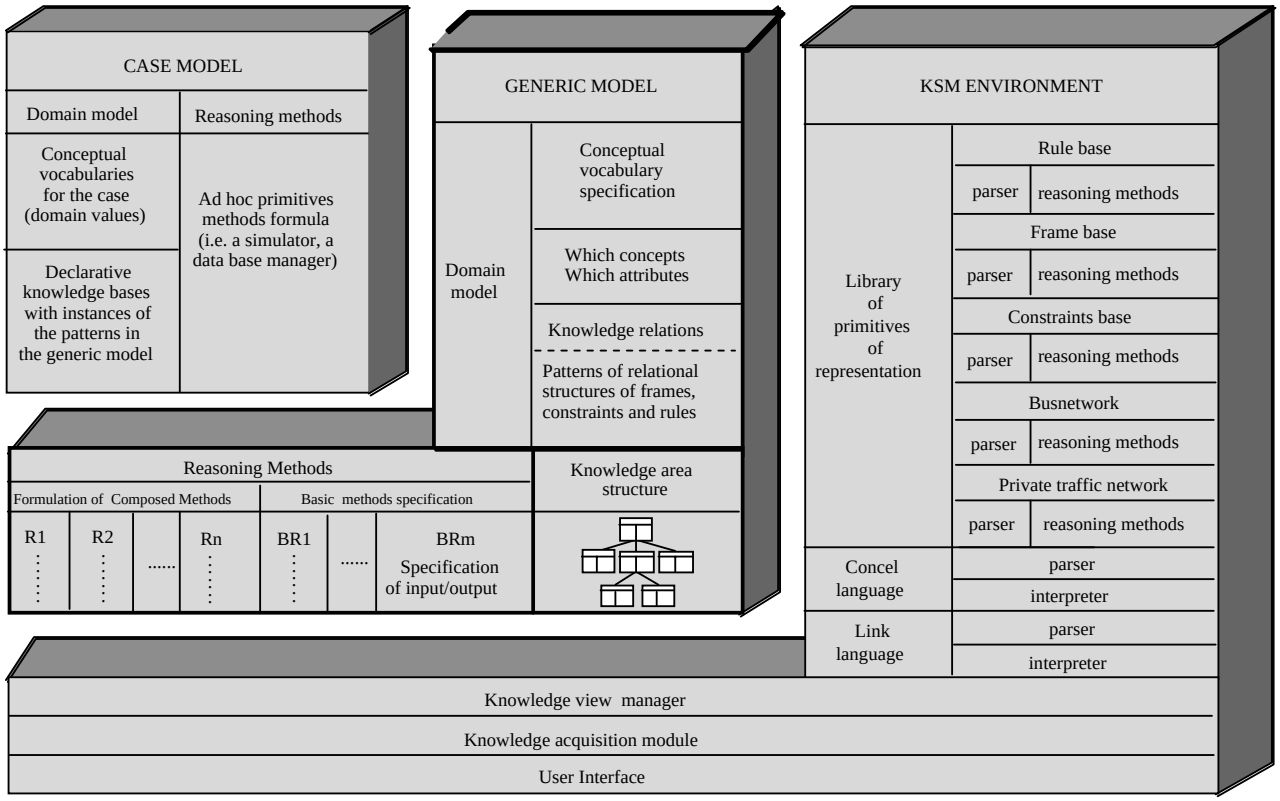

Figure 18: The three main components of an application deveoped using KSM